UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

TCPSnitch

Dissecting the Socket API Usage

Dissertation presented by
Gregory VANDER SCHUEREN

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Olivier BONAVENTURE

Reader(s)
Quentin DE CONINCK, **Gregory DETAL**

Academic year 2016-2017

Abstract

Networked applications interact with the TCP/IP stack through the socket API. Over the years, the computing environment in which this API is being used has dramatically changed and many extensions have been added. In this thesis, we propose and implement `TCPSnitch`, an open-source software that collects detailed traces of the interactions between networked applications and the TCP/IP stack and sends them to a publicly available database, `tcpsnitch.org`, exposing various statistics. We collect about 5 M API calls made by more than 130 Linux and Android applications on 24 K sockets. Our analysis reveals that the socket API usage in the wild differs greatly from textbook usage. Many Internet sockets do not exchange data, many API calls are redundant and many calls use tiny data buffers. On Android, UDP sockets are mainly used as a shortcut to retrieve information about the network configuration using `ioctl()` and most applications use various socket options even if the Java API does not expose them directly. `TCPSnitch` and the associated dataset are publicly available.

Acknowledgments

I am grateful to Olivier Bonaventure, my supervisor, for proposing me this thesis subject. Back in March 2016, I remember asking for a subject related to operating systems and networking that involves a lot of programming. This is exactly what I got. I believe this was key to keep my highly motivated throughout the year.

I would also like to thank Olivier Bonaventure and Quentin De Coninck for their availability and support throughout the year. We met almost every week since September 2016 and they provided me with invaluable inputs, feedback and ideas for new directions.

I am also thankful to Quentin De Coninck and Gregory Detal for accepting the task of being my readers.

Finally, I owe a special thank to my parents, who always supported me when I decided to go back to school to start this second Master's degree in Computer Science.

Contents

Abbreviations and symbols

API	Application Programming Interface
ATM	Asynchronous Transfer Mode
BPF	Berkeley Packet Filter
BSD	Berkeley Software Distribution
C	The C programming language
CDF	Cumulative Distribution Function
CPU	Central Processing Unit
DNS	Domain Name System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP over Transport Layer Security
I/O	Input/Output
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
JSON	JavaScript Object Notation
LAN	Local Area Network
LTE	Long-Term Evolution
MB	Megabyte
OS	Operating System
OSI	Open Systems Interconnection
POSIX	Portable Operating System Interface
QUIC	Quick UDP Internet Connections
SDK	Software Development Kit
SHA-1	Secure Hash Algorithm 1
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URL	Uniform Resource Locator
eBPF	Extended BPF

Chapter 1

Introduction

Computer applications communicate over a computer network using *network protocols*. These network protocols define the rules and conventions for their interactions. In practice, multiple layers of network protocols are usually involved and each protocol is typically designed to solve a particular problem. For instance, when a web client communicates with a web server, both applications use the Hypertext Transfer Protocol (HTTP) to exchange hypertext objects. These HTTP messages are themselves carried by the Transmission Control Protocol (TCP), whose main purpose is to provide a reliable byte-stream channel. In turn, the Internet Protocol (IP) carries these TCP messages and takes care of routing packets and addressing hosts on the network.

The *Internet protocol suite*, commonly known as TCP/IP, includes all the dominant network protocols used over the Internet nowadays. A *TCP/IP stack* is a «stack» of programs that implement a complete set of network protocols from the Internet protocol suite. Fortunately for application developers, a TCP/IP stack is typically provided by the operating system or an existing library. This allows processes to communicate over a computer network with minimal effort. In the OSI model of computer networking, the TCP/IP stack takes care of the protocols at the transport layer and below. In practice, this means that the TCP/IP stack typically offers the services of a transport protocol to the application, such as UDP or TCP, and the lower layer protocols remain a black box for the application.

The *socket API* is the standard application programming interface for communicating with a TCP/IP stack. This API was introduced together with the 4.2 release of the BSD Unix distribution that included a functional TCP/IP stack [24]. When the socket API was designed, TCP/IP was one family of network protocols among many others and it was important to abstract those protocol families. The core concept of a *socket* is an abstraction

used to represent one of the endpoints of a network communication channel. Through a socket, an application may send data which can be received by the application connected to the other endpoint of the channel. Concretely, a socket is represented to user-space applications by a file descriptor, following the «everything is a file» philosophy of Unix. Much like a file, an application may open and close a socket, read and write to a socket. The heart of the socket API is a set of basic system calls for operating sockets, including `socket`, `bind`, `connect`, `accept`, `close`, `listen`, `send`, `receive`, etc. Those system calls interact with the underlying network implementation that is part of the operating system kernel. The socket API was not the only approach to interact with the network stack. The STREAM API, based on [26] was extended to support the TCP/IP protocol stack and used in Unix System V [25].

Over the years, the popularity of the socket API grew in parallel with the deployment of the global Internet. Nowadays, many consider the socket API as the standard API for simple networked applications and the socket API has evolved as a component of the POSIX specification. Several popular textbooks are entirely devoted to this API [30, 10]. Given the importance of web-based applications, many developers do not interact directly with the socket API anymore but rely on higher-level abstractions. For instance, programming languages such as Java or Python include libraries exposing URL and implementations of HTTP/HTTPS. For C developers, libraries such as `libcurl` also provide higher-level abstractions.

During the last 30 years, the socket API has evolved, with new features added over the years. Some were dedicated to the support of specific features, such as ATM [4] or Quality of Service [3]. Other extensions [23, 13] focused on improving the interactions between applications and the underlying stack through `select`, `poll`, `epoll`, etc. On Linux, new system calls to directly send pages or entire files (like `sendfile`) were added. Furthermore, socket extensions have been defined for each new transport protocol [27, 21]. Socket extensions have also been proposed to deal with multihoming [28] and specific APIs have been implemented on top of Multipath TCP [16, 17].

Recently, the Internet Engineering Task Force created the Transport Services (taps) working group whose main objective is *to help application and network stack programmers by describing an (abstract) interface for applications to make use of Transport Services*. Although this work will focus on abstract transport services, understanding how the current APIs are used by existing applications will help in designing generic transport services that correspond to their needs.

Since the inception of the socket API in the early 1980s, the computing environment in which this API is being used has dramatically changed. This same socket API is now used on smartphones, tablets and embedded

devices that did not exist when it was first created. Applications interact with numerous layers of software frameworks and libraries which sit on top of the OS. This raises many questions. Is the socket API still suited to support modern workloads? Are its abstractions used the way they were intended to be used? Did some parts of the socket API become obsolete over time? To the best of our knowledge, empirical data lacks to answer these questions and little work has been devoted to investigating how modern applications make use of the socket API. This work intends to start filling this gap.

We propose and implement `TCPSnitch`, an open-source software that collects detailed traces of the interactions between networked applications and the TCP/IP stack and sends them to a publicly available database, `tcpsnitch.org`, exposing various statistics. Our objective is to provide empirical data about the socket API usage by modern applications. This work is targeted to both researchers, software developers and standard bodies. We also believe that `TCPSnitch` has educational value and could be used in classes that teach network programming.

The rest of this document is structured as follows:

- Chapter 2 provides background information needed to understand the rest of this document and reviews existing work.
- Chapter 3 investigates the inner-workings of `TCPSnitch` and `tcpsnitch.org`.
- Chapter 4 describes the public dataset exposed by `tcpsnitch.org`, which is analyzed in the remaining chapters.
- Chapter 5 dissects how applications use UDP sockets.
- Chapter 6 analyzes the socket API usage on TCP sockets.
- Chapter 7 summarizes our main findings and contributions and gives directions for future work.

Chapter 2

State-of-the-art

The previous chapter introduced `TCPSnitch`, our open-source software that collects detailed traces of the interactions between networked applications and the TCP/IP stack. This chapter lays the groundwork before delving into the details of `TCPSnitch` in the next chapter.

Different solutions have been proposed and implemented to analyze the utilization of system and library calls by applications. Two approaches are possible. The static approach analyzes the application code (binary or sometimes source for open-source applications) and extracts the interesting calls from the corresponding files. The dynamic approach instruments the application and intercepts the system or library calls. `TCPSnitch` follows the latter approach. This chapter contrasts both approaches and reviews the related work and existing tools.

Section 2.1 is a refresher on important concepts used in the rest of this document. Section 2.2 reviews the related work and the existing tools. It also contrasts the pros and cons of the static and dynamic approaches. Finally, section 2.3 explains how to implement a dynamic tracing tool such as `TCPSnitch` on a Linux machine using the `LD_PRELOAD` environment variable.

2.1 Background

This section introduces some fundamental concepts related to shared libraries and the dynamic linker. It also briefly explains what the C standard library is and how programs use it to execute system calls. For additional information about these topics, the interested reader can consult [18, 31, 6], which were used to document this section.

2.1.1 Shared libraries and the dynamic linker

When a program is compiled, the compiler converts source files into *object files*. An object file contains machine instructions that correspond to the instructions in the source program. The *linker* serves to package multiple object files into a single executable or library, a process known as linking. Fig. 2.1 illustrates how source files are turned into an executable after the compiling and linking phases.

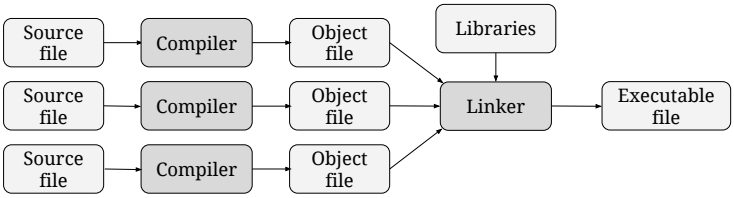

Figure 2.1: Compiling and linking - The compiler converts source files into object files (compiling). The linker packages multiple object files into a single executable or library (linking).

Programming *libraries* are object files that package together commonly used pieces of code. They allow for code reuse as multiple programs can be linked against the same library. There are two ways to link a program with a library: *static linking* and *dynamic linking*. With static linking, external symbols contained in the library are resolved at compile-time and a copy of the library code must be embedded in the final executable. With dynamic linking, the external symbols are resolved at load time[1]. At compilation time, the linker merely adds the name of the library to the list of dependencies of the program. The code of the library is not included in the final executable. *Shared libraries* are library files designed to be dynamically linked against programs at load time. When multiple processes depending on the same shared library run concurrently, a single copy of the library is loaded into memory and shared by all processes.

When a process starts executing, it is the role of the *dynamic linker* to load and link the needed shared libraries. When binding symbols at load-time, the dynamic linker searches for symbols through the libraries in the same order as they were linked against the program. If multiple libraries define the same symbol name, the definition in the library that was first loaded will apply.

[1] To be precise, symbols are often lazily resolved at run-time. However, this optimization can be ignored for our discussion.

2.1.2 System calls and library functions

System calls allow a process to request the kernel of the operating system to perform a service on its behalf. For instance, a process may execute a system call to read data from disk or to create and execute a new process. Together, the system calls form the interface between the processes and the operating system and are the entry-points to the kernel code.

To invoke a system call, a process must first place its arguments in specific CPU registers and on the stack. In particular, it must place the system call number to identify which system call should be executed. The process then invokes a trap[2] machine instruction which switches the processor from user mode to kernel mode. In response to this trap instruction, the kernel executes its system call handling routine. When the execution of the system call terminates, the kernel places the system call return value on the stack and returns the processor to user mode.

Making system calls is tedious and non-portable because it is architecture-specific. Fortunately, the standard C library (libc) supplements a wrapper function to most system calls on Unix systems. These wrapper functions offer a programmer-friendly interface to execute the system call. For instance, libc exposes the `open()` function which executes the `sys_open` system call. From the user process perspective, invoking a system call becomes similar to calling a regular C function. In turn, this function invokes the appropriate kernel system call taking care of the low-level details.

Note that programs can either make the system call directly or call the wrapper library function. Fig. 2.2 highlights this point and summarizes the concepts introduced in this section.

2.2 Related work and tools

The socket API was born in the early 1980s and is ubiquitous on modern computers. To the best of our knowledge, there has been surprisingly little empirical data published about the socket API usage in the scientific literature. This section reviews the related work and some related existing tools.

2.2.1 Static analysis

Several researchers adopted the static analysis approach to study networked applications. In 2011, Komu et al. [19] analyzed the source code of

[2] A trap is a software-generated interrupt.

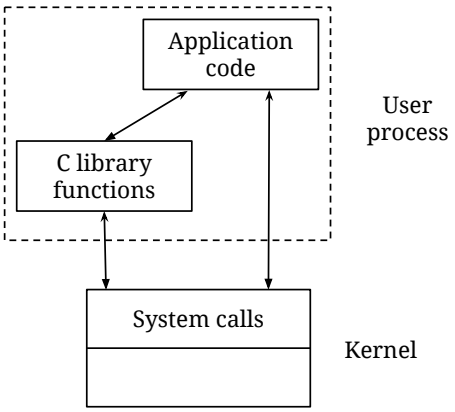

Figure 2.2: System calls and C library functions [**31**] - On the left path, the application executes the system call via the C library function. On the right path, the application invokes the system call directly.

1	IPv4-IPv6 hybrids	26.9%
2	TCP-UDP hybrids	26.3%
3	Obsolete DNS resolver	43.3%
4	UDP-based apps with multihoming issue	45.7%
5	Customize networking stack	51.4%

Table 2.1: Core socket API findings [19] - The authors identify five behaviors and quantify the proportion of applications displaying each behavior.

2187 Ubuntu C packages and 4 network frameworks to detect the presence of certain keywords of the socket API. They characterized statistically some aspects of the socket API usage and investigated for programming pitfalls. The authors concluded their paper with a list of five findings about the core socket API usage (see Table 2.1).

In 2016, Tsai et al. [32] disassembled binaries of 30 K Linux applications using `objdump` and performed a call-graph analysis to study the Linux API usage. After extracting an API footprint for each application, the authors weighted these footprints by the package popularity to estimate the real API usage. They showed that a substantial part of the Linux API is unused in practice.

Still in 2016, Atlikadis et al. [5] were first to study the POSIX API usage, which includes the socket API. The authors proposed `libtrack`, a tool which can trace the usage of a given native C library by applications on Ubuntu, macOS and Android. Via static analysis, they inspected 1.1 M Android applications and about 72 K Ubuntu packages for linkage with

POSIX functions.[3]

The main advantage of such static approaches is that it is possible to analyze a large number of applications to determine the system calls and library functions used by the majority of the applications. Unfortunately, it is very difficult to establish which parameters are passed to these functions or how frequently they are called. Source code analysis is also impractical for closed-source applications.

2.2.2 Dynamic analysis

As early as 1985, Ousterhout et al. [22] adopted a dynamic approach to conduct a trace-driven analysis of the Unix 4.2 BSD File System. Coincidentally, in was in this same 4.2 BSD release that the original socket API was born. The authors instrumented the 4.2 BSD system to collect information about file access and recorded user-level activity in trace files.

The `libtrack` tool proposed by Atlikadis et al. [5] also supports dynamic tracing of functions invocations. Via dynamic analysis, they analyzed the POSIX API usage by 100 applications: 45 on Android, 10 on macOS and 45 on Ubuntu. An advantage of `libtrack` is that it intercepts more than a thousand POSIX functions in a generic manner. However, this genericity also means that `libtrack` is not tailored to the peculiarities of individual functions: `libtrack` does not track functions arguments and return values. Their first finding is that high-level frameworks drive the POSIX API usage: modern applications are no longer written directly atop the POSIX API, but rely on platform-specific frameworks and libraries. Second, they observed that `ioctl()` dominates the modern POSIX usage patterns and serves to build new abstractions missing from the POSIX standard. They also found that the same new abstractions are arising across these three OSes, because of the same limitations. They claim that these new abstractions are not converging on any new standard.

On Unix variants, existing tools such as `strace` or `ltrace` can also be used to collect traces of system and library calls respectively. However, both tools are not dedicated to the inspection of Internet sockets usage and have shortcomings for our purpose. First, they fail to group system calls per socket. Second, although the `-e trace=network` option of `strace` may restrict the output to network related system calls, `ltrace` (which works at the socket API level) does not support this feature. Finally, the output structure of both utilities is not designed to be easily processed by

[3] We mention the main results in the next section as they seem to originate mostly from the dynamic analysis module of `libtrack`.

a computer program. While we could overcome these shortcomings with further processing, both tools seem not specialized enough for our purpose.

Aside from existing ready-to-use applications, multiple tools and technologies could be used to create an application that traces the socket API usage. While [14] lists many of these tools, we only describe two of the most serious contenders: eBPF and `ptrace`.

The Extended Berkeley Packet Filtering (eBPF) framework is an in-kernel virtual machine that allows executing custom eBPF programs directly into the Linux kernel. We can attach these eBPF programs to different events in the kernel such as the arrival of a network packet or the execution of a given function in the kernel code. A user-space process can then communicate with these eBPF programs to retrieve information from the kernel. This solution has the advantage of being extremely flexible. It allows retrieving information from the kernel that could not be accessed otherwise. It also has a low performance overhead as these eBPF programs run directly into the kernel. An inconvenience is that this technology is relatively new and Linux specific.

Another serious contender is `ptrace`, a system call available on most Unix platforms that allows one process to observe and control the execution of a second process. This system call is very useful to perform system call tracing and both `strace` and `ltrace` use the `ptrace` system call under the hood [8, 7]. A drawback is that the behavior of `ptrace` differs significantly depending on the operating system and the architecture of the machine [20].

Another alternative is to modify the behavior of the Linux dynamic linker using the `LD_PRELOAD` environment variable. This trick allows injecting a custom shared library into an executable to intercept library calls. The first advantage of this solution is its portability: it can be used on any platform where the dynamic linker exposes a similar feature. Another advantage is its simplicity as it only requires to write a regular C library. A drawback of this approach is that it does not support some corner cases: programs that are statically linked against libc and programs that execute system calls directly without using the libc wrapper functions. Fortunately, these are the exception rather than the rule and this approach will be compatible with most applications of interest. This is the approach we have chosen. The next section explains in detail how to build a tracing tool on top of `LD_PRELOAD`.

Compared to the static approach, the main advantage the dynamic approach is being able to trace sequences of calls and collect information about the function parameters and the return values. This enables to observe how and when socket API calls and options are used, the size of the buffers used by `send()/recv()`, which thread called the API function, etc. While static

analysis tools such as [19] or [32] give indications about the possible usage of some socket API interfaces or options, a dynamic tool allows observing their actual use and detecting patterns of interactions. A disadvantage is that a dynamic analysis only traces the scenario actually executed on the traced application, but not all code paths.

2.3 LD_PRELOAD in action

As shown in Fig. 2.3 adapted from [31], `TCPSnitch` uses `LD_PRELOAD` to intercept the libc functions calling the system calls. This section explains how we can modify the Linux dynamic linker behavior using the `LD_PRELOAD` environment variable. It also shows how injecting a custom shared library into an executable allows capturing the API calls made by applications.

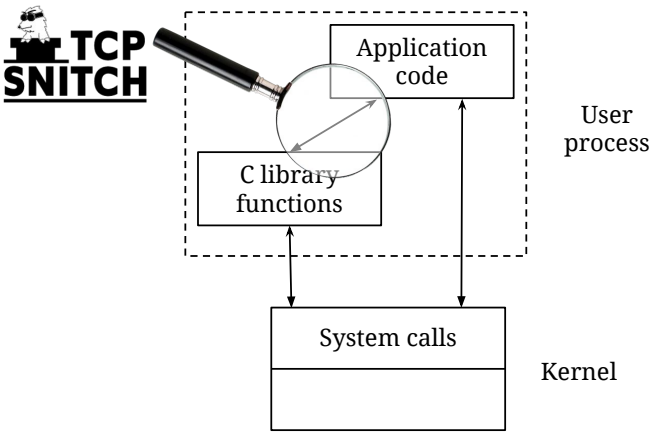

Figure 2.3: Interception of library calls by `TCPSnitch` - `TCPSnitch` does not intercept system calls invoked without libc.

An interesting feature of the Linux dynamic linker is the ability to link user-specified shared libraries *before* the regular dependencies of a program. The `LD_PRELOAD` environment variable controls this feature and may contain a list of user-specified libraries. In particular, this `LD_PRELOAD` variable may force the dynamic linker to link a user-specified shared library *before* the libc library. As a result, any function defined in this user-specified library would take precedence over a function with the same signature defined in libc. This trick allows intercepting calls to libc functions using a custom shared library that redefines the functions of interest. Such a shim library can transparently intercept the calls without perturbing the traced application by calling the original libc functions after performing its own processing.

We can use the `dlopen API` of the dynamic linker to call the original

libc function. In particular, this API exposes the `dlsym()` function whose purpose is to search in open libraries for a given symbol and return its address. `dlsym()` accepts two arguments, a handle to a dynamically loaded shared library and a symbol name, and returns the address of such symbol if it was found in the library pointed by the handle (or in its dependency tree[4]). `dlsym()` also supports *pseudo-handles*, such as `RTLD_NEXT`, which instructs the dynamic linker to search for the given symbol name in all the shared libraries loaded after the one invoking the `dlsym()` function.

The following three code snippets provide a concrete example of using the `LD_PRELOAD` environment variable to intercept a libc function. First, code snippet 2.1[5] shows a minimal shared library that intercepts the `puts()` libc function. For each intercepted call, it simply logs a greeting message and then calls the original libc `puts()` function. Then, code snippet 2.2 shows a small C program that calls once the `puts()` function. Finally, the code snippet 2.3 shows how to wire everything together. It first compiles the `hello.c` program and the `lib.c` shared library. Then, it executes the `hello.out` program with `LD_PRELOAD` pointing to our minimal shared library. The output of `hello.out` shows that our small shared library correctly intercepted the call to `puts()`.

```c
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>

typedef int (*orig_puts_type)(const char *s);
orig_puts_type orig_puts;

int puts(const char *s) {
    if (!orig_puts)
        orig_puts = (orig_puts_type)dlsym(RTLD_NEXT, "puts");
    // Perform custom processing here...
    orig_puts("Hello from shim library!");
    return orig_puts(s);
}
```

Listing 2.1: lib.c - A minimal shared library that intercepts `puts()`, logs a greeting message and calls the original `puts()` implementation.

```c
#include <stdio.h>
#include <string.h>
int main(void) { puts("Hello!"); }
```

Listing 2.2: hello.c - A small program that calls `puts()`.

[4] A library may depend on other libraries.

[5] We omit error checking code for brevity. The attentive reader may have noticed that calling `puts()` instead of `orig_puts()` for displaying the greeting message would result in an infinite loop.

```
1 $ gcc hello.c -o hello.out
2 $ gcc -fPIC -shared -o lib.so lib.c -ldl
3 $ LD_PRELOAD=./lib.so hello.out
4 Hello from shim library!
5 Hello!
```

Listing 2.3: LD_PRELOAD in action - The program `hello.c` and the shared library `lib.c` are compiled. The program is then executed with `LD_PRELOAD` pointing to the shared library. The output of the program shows that the shared library intercepted the call to `puts()`.

2.4 Conclusion

This chapter laid the foundations to read the rest of this document. It started with a refresher on some essential concepts for understanding the next chapters. We then explained that two approaches are possible to analyze the utilization of system and library calls by applications: static and dynamic analysis. `TCPSnitch` follows the latter approach.

We reviewed existing work and tools based on both approaches. A major benefit of dynamic analysis is the ability to observe the actual usage of functions rather than deducing their probable use. It allows observing sequences of calls, detecting patterns of interactions and collecting information about the functions parameters and their return values.

We explained how `TCPSnitch` can use `LD_PRELOAD` to trace libc function calls. The next chapter describes `TCPSnitch` itself. It explains the overall architecture of the program and digs into the implementation of both `TCPSnitch` and `tcpsnitch.org`.

Chapter 3

TCPSnitch

In the previous chapter, we saw the benefits of using a dynamic approach to analyze the utilization of system and library calls by applications. We also showed how to build a minimal shared library to intercept libc functions calls with `LD_PRELOAD`. This chapter builds on this and gives an in-depth description of `TCPSnitch` and `tcpsnitch.org`.

We explain the overall architecture of both tools and detail the main software tools used for their development. When deemed necessary, we also delve into some important implementation details. The goal of this chapter is to provide enough information to anyone interested in contributing to this project. After reading this chapter, the reader should feel confident about browsing the source code to modify or extend `TCPSnitch`'s behavior.

Section 3.1 gives a bird-eye view of `TCPSnitch` and `tcpsnitch.org`. Section 3.2 delves into the implementation details of `TCPSnitch` while section 3.3 covers the architecture of the `tcpsnitch.org` platform.

3.1 Overview

We propose and implement `TCPSnitch`, an open-source software that collects detailed traces of the interactions between networked applications and the TCP/IP stack and sends them to a publicly available database exposing various statistics. `TCPSnitch` currently intercepts 40 functions related to the network stack[1] and runs on Linux and Android. Compared to simpler tools like `strace` and `libtrack`, a major benefit of `TCPSnitch` is that the collected data is uploaded to a public database. The web interface of this

[1] Appendix A.1 lists these 40 functions.

database, available on `tcpsnitch.org`, provides different visualizations of the database and allows users to browse through the collected traces.

In practice, `TCPSnitch` runs the specified command until it exits, much like `tcpdump`. For instance, one may simply execute "`tcpsnitch curl uclouvain.be`" to trace the `curl` program. `TCPSnitch` tracks the functions that are applied on each Internet socket with their timestamp, parameters and return value. For each socket, `TCPSnitch` builds an ordered list of function calls. Socket traces are then written to text files where each line is a JSON object representing a single function call. For instance, code snippet 3.1 shows how a `connect()` function call might look like in a socket trace. We observe that `TCPSnitch` also captures the thread id when intercepting a function call, which allows observing how programmers use threads to manipulate sockets.

```
{
    "type": "connect",
    "timestamp_usec": 1491043720731853,
    "return_value": 0,
    "success": true,
    "thread_id": 17313,
    "details": {
        "addr": {
            "sa_family": "AF_INET",
            "ip": "127.0.1.1",
            "port": "53"
        }
    }
}
```

Listing 3.1: JSON representation of a `connect()` call - `TCPSnitch` captures the return value, the arguments and additional information such as the timestamp or the thread id.

As `TCPSnitch` uses `LD_PRELOAD` under the hood, it is able to follow forks and group traces by process. `TCPSnitch` packages the multiple socket traces into an application trace which also includes system information, such as the network configuration or the kernel version. `TCPSnitch` also supports additional features such as the ability to extract the `TCP_INFO` socket option at user-defined intervals or to record a `.pcap` packet trace for each individual socket.

Because `TCPSnitch` traces contain lots of information to process, we also propose `tcpsnitch.org`, an open-source platform designed to centralize, visualize and analyze the traces. By default, traces are automatically uploaded to `tcpsnitch.org`. Upon uploading, multiple metrics and charts are computed on the trace to get a better understanding of the network footprint of the application. For instance, `https://tcpsnitch.org/app_traces/138` dissects the network footprint of the Facebook application on Android. Users

can browse the traces and visualize statistics about the dataset or a particular trace. Our hope is that this platform will enable the community to get a better understanding of networked application behaviors.

To preserve the user privacy, he/she can opt-out for the collection of sensitive metadata. `TCPSnitch` does not trace the utilization of the DNS libraries and thus does not collect domain names. With `send()/recv()`, `TCPSnitch` only collects the buffer pointers and sizes, not the actual payload. Furthermore, IP addresses that are collected as parameters of the traced function calls are replaced by the low order 32 (resp. 128) bits of a SHA-1 hash computed over the concatenation of a random number generated by `TCPSnitch` when it starts and the IP address. Note that this anonymization method is not prefix-preserving. While the loss of the prefix relationships between addresses could render the traces unusable in other studies (e.g. routing performance analysis or clustering of end-systems) [11], it is immaterial for our purpose. As suggested by [12], we leave intact addresses that fall inside the loopback and link-local address ranges. If the anonymization process yields an address that lands in those special ranges, we repeat the process until it lands outside these ranges.

3.2 Implementation of `TCPSnitch`

`TCPSnitch` is actually composed of two components: the `tcpsnitch` executable itself and the `libtcpsnitch.so` shared library. The shared library is written in C and counts about 6500 lines of code, without blanks and comments. The `tcpsnitch` executable is a shell script of about 350 lines of code, without blanks and comments.

The purpose of the `tcpsnitch` shell script is to provide a friendly user interface to link an executable with the `libtcpsnitch.so` shared library. It parses the options and arguments given by the user and performs basic validations on the inputs. It is also in charge of starting the traced application in a modified environment: it makes the `LD_PRELOAD` environment variable point to `libtcpsnitch.so` and saves the option values in environment variables such that `libtcpsnitch.so` is later able to retrieve them.

The `libtcpsnitch.so` shared library performs the actual tracing task and is described in depth in the following sections. It depends on two other libraries: the `jansson`[2] library for encoding JSON data and the `pcap`[3] library for capturing `pcap` packet traces.

Before going further, we define the term *event* which will serve for the rest

[2] `www.digip.org/jansson`
[3] `www.tcpdump.org`

of our discussion. An *event* corresponds to an entry in the JSON trace of a socket. Most of the time, this corresponds to a function call on the socket but other types of events might appear in traces. For instance, `TcpInfo` events are inserted in traces when `TCPSnitch` is requested to capture this socket option at regular intervals. Another example is the `ForkedSocket` event. This event is inserted as the first event in the traces of sockets inherited after a `fork()` call. These `ForkedSocket` events serve to make each trace self-contained: these events contain information such as the socket domain or the socket type, which would otherwise be missing from the trace.

3.2.1 Modules overview

Before delving into the code, we describe the core modules of the `libtcpsnitch.so` shared library and explain their purpose. By module, we mean a pair of `.h` and `.c` files declaring and implementing a group of related features. Following the logical flow of the program, these modules are:

`libc_overrides`: Redefines the intercepted libc functions and acts as the entry-point to the library code.

`sock_events`: Exposes *event handlers*, functions used to register events.

`init`: Initializes the library.

`json_builder`: Builds the JSON representation of the events.

In addition to these central modules, there are a few auxiliary modules:

`resizable_array`: Implements a map data structure which backs the `tcp_events` module.

`lib`: Contains generic purpose functions.

`logger`: Contains logging and debugging functions.

`string_builders`: Contains helper functions to build string representations for various data types.

`constants`: Contains helper functions to convert C integer constants to a string representation.

`packet_sniffer`: Controls the capture of `pcap` traces.

Fig. 3.1 summarizes the interactions between these components. Arrows point from a component that calls functions defined by another component. Since all components use the `lib` and `logger` components, arrows are omitted for clarity. The vertical dashed arrows depict the interception of the libc functions called by the traced application. All components also call functions defined in libc and these arrows are also omitted.

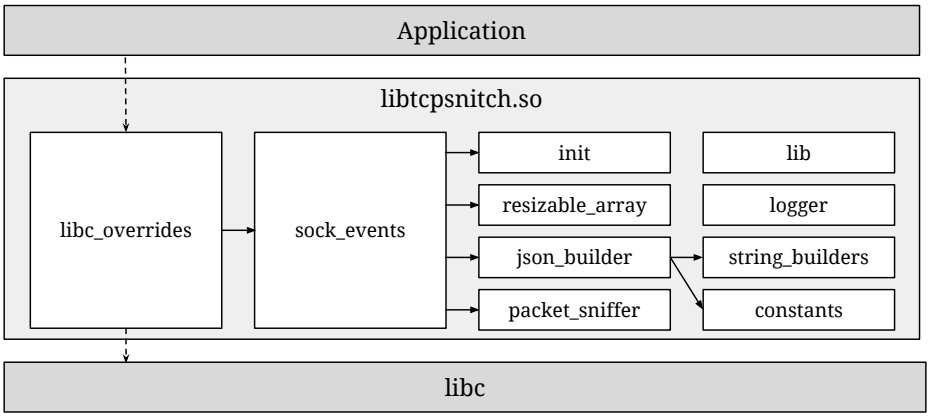

Figure 3.1: TCPSnitch architecture - An arrow points from a component that calls functions defined by another component.

3.2.2 Module `libc_overrides`

This module redefines all the libc functions that we wish to intercept. As such, it is the entry point to the library code. For each intercepted function, the code performs roughly the same tasks:

1. Retrieve a reference to the original libc implementation using the `dlsym()` function of the dynamic linker API.
2. Call the original libc function with the same arguments and save both the return value and the value of `errno`.
3. If the file descriptor corresponds to an Internet socket, call the corresponding event handler from the `tcp_event` module, passing along the parameters, the return value and the value of `errno`.
4. Restore `errno` to its previous value and return the return value of the original libc function.

Code snippet 3.2 shows how the `connect()` function is overridden.

```
1  typedef int (*connect_type)(int fd, const struct sockaddr *addr,
2                              socklen_t n);
3  connect_type orig_connect;
4
5  int connect(int fd, const struct sockaddr *add, socklen_t n) {
6      if (!orig_connect)
7          orig_connect = (connect_type)dlsym(RTLD_NEXT,"connect");
8
9      int ret = orig_connect(fd, add, n);
10     int err = errno;
11
12     if (is_inet_sock(fd)) sock_ev_connect(fd, ret, err, add, n);
13
```

```
14      errno = err;
15      return ret;
16  }
```

Listing 3.2: Override of connect() - The overridden function calls the original libc implementation and saves both the return value and the value of `errno`. If the file descriptor is an Internet socket, it calls the corresponding event handler. Finally, it restores `errno` and returns the saved return value.

3.2.3 Module `init`

Usually, the `socket()` function in `libc_overrides` is the very first function of the library to be called.[4] If the file descriptor is an Internet socket, `socket()` calls `tcp_ev_socket()` in `sock_events`, which immediately calls the `init_tcpsnitch()` function exposed by the `init` module. This function performs the initialization of the library. Note that the body of `init_tcpsnitch()` is executed at most once per process: it is protected by a mutex and a variable serves to remember if it has been initialized. During this first execution, it performs the following tasks:

1. Open two new streams to known file descriptors that will serve as the `stdout` and `stderr` of the library. This allows the wrapper shell script to distinguish the standard streams of the traced process from the standard streams of the library to perform filtering operations.
2. Retrieve the option values stored in the environment.
3. Create the directory for storing the traces.
4. Configure the `logger` module based on the option values.
5. Start a new thread which periodically dumps JSON data to files.

The `init` module also exposes a `reset_tcpsnitch()` function which resets the library after a `fork()` call. For instance, this involves marking the forked sockets and creating a new directory for writing the traces of the new process.

3.2.4 Module `sock_events`

The `sock_events` module is the core of the library. In addition to declaring structures for holding the state of a socket and its events, it exposes event handlers to register intercepted libc functions. For instance, it exposes the `sock_ev_connect()` event handler whose declaration is shown in code snippet 3.3. Most event handlers accept the same first three parameters while the other parameters are specific to the intercepted libc function.

[4] This is not always the case however. For instance, a forked process that inherits open sockets will not necessarily call `socket()` first.

```
void sock_ev_connect(int fd, int ret, int err,
                     const struct sockaddr *addr, socklen_t len);
```

Listing 3.3: An event handler declaration - Most event handlers accept the same first three parameters while the other parameters are specific to the intercepted libc function.

An event handler starts by retrieving the `Socket` structure associated with the file descriptor. The mapping between file descriptors and `Socket` structures is maintained in a map data structure, whose implementation is detailed in the next section. The main purpose of a `Socket` structure is to keep a linked list of events for a given socket and to store bookkeeping data such as the number of bytes sent on the socket.

After retrieving the `Socket` structure, the event handler allocates and fills the corresponding event structure. The first element of an event structure is always a generic `SockEvent` structure. In a sense, they extend the parent structure `SockEvent`: generic event information is stored in the `SockEvent` structure while event-specific information is stored in the child structure. This design allows simulating polymorphic functions by accepting a generic `SockEvent` structure as argument. Code snippet 3.4 shows how the `SockEvRecv` structure extends the `SockEvent` structure.

```
typedef struct {
        SockEventType type;
        unsigned long timestamp_usec;
        int return_value;
        bool success;
        int err;
        long id;
        pid_t thread_id;
} SockEvent;

typedef struct {
        SockEvent super;
        size_t bytes;
        int flags;
} SockEvRecv;
```

Listing 3.4: The `SockEvRecv` structure extends the `SockEv` structure - We simulate polymorphic functions by accepting a generic `SockEvent` structure as argument.

3.2.5 Module `resizable_array`

The `sock_events` module uses a map data structure to maintain the mapping between file descriptors and `Socket` structures. In this map, keys are file descriptors while values are pointers to `Socket` structures. Under the hood, this data structure is implemented as a resizable array and the file descriptor gives the position of an element in the array.

The vast majority of operations on the map are read operations. For each intercepted libc function call, we must retrieve the `Socket` pointer associated with the file descriptor. On rare occasions, we also write to the map: when a socket is opened or closed and when the underlying array is being resized.

An array has the advantage of offering constant time read operations. Since a read operation is performed for each intercepted call, it must better be extremely fast. Another advantage is simplicity: we start with a native C array of fixed size and double its size when we need to store an element past its current bound.

Because we index `Socket` structures in the array by their corresponding file descriptor number, the array will often be sparse. In the space of opened file descriptors, many may not be Internet sockets. With f being the number of open file descriptors, and s being the number of opened Internet sockets, the array uses $O(f)$ space while it should ideally consume $O(s)$ space. We claim that this space consumption is negligible. As the array consumes 8 bytes[5] for each element, it would in the worst-case take about 1 MB of memory if there were 2^{16} open file descriptors.[6]

3.2.6 Concurrency issues

The resizable array is an instance of the readers-writers problem. Multiple threads can safely read concurrently from the array. However, when a single thread starts writing to the array we must ensure that no other thread is accessing the same portion of the array. Because reads are much more frequent that writes, we use a single read-write lock[7] for synchronizing threads accessing the array. This solution offers a high degree of parallelism for reading and simplicity when locking the array for writing. Allowing at most a single writer to access the array concurrently is perfectly acceptable since writes are occasional.

Concurrency issues also arise when multiple threads are manipulating a single `Socket` structure. In this case, writes are the norm: a thread retrieves a `Socket` pointer from the array, appends a new `SockEvent` to its list of events and updates bookkeeping information on the `Socket` structure itself. We use coarse-grained locking on the `Socket`: the main structure, along with its linked-list of events, is protected by a single mutex.

[5] A pointer is 8 bytes long on a 64-bit machine.

[6] About $8 * 2^{16} * 2$, where the last 2 factor accounts for the array doubling its size when the `Socket` pointer of the highest numbered file descriptor is inserted.

[7] A read-write lock can be in three modes: locked in read mode, locked in write mode and unlocked. At any time, only a single thread can hold the lock in write-mode while multiple threads can hold the lock in read-mode.

Since `Socket` structures are exclusively accessed via the map structure, a convenient solution to solve our synchronization issues is to make the map in charge of locking the individual `Socket` structures. This has the advantage of hiding the locking logic in the map implementation. When an item is retrieved from the map, it is locked before being returned. The item is then considered in use by the calling thread until it explicitly releases it. Once the item is released, the calling thread may no longer use it. Because the array is sparse, it would be wasteful to allocate a mutex for each position in the array. Instead, an item is wrapped into an `ElemWrapper` structure before being inserted. A wrapper structure only holds two elements: the item itself and a mutex for synchronizing threads accessing the item.

Putting it all together, code snippet 3.5 illustrates simplified methods for operating the map. The attentive reader might notice that the read-write lock is not immediately released after retrieving an item from the map with `get()`. This simplifies the deletion of items. By unlocking the read-write lock in the `release()` method, the read-write lock keeps count of the valid item references in the program. Before removing a `Socket` structure from the map to deallocate it, we lock the read-write lock in *write-mode*. This guarantees that no thread currently holds a valid reference to the `Socket` structure (or any other such structure). It also guarantees that no other thread can get such a reference since the read-write lock is exclusive in write mode. Note that it is because deletions are infrequent that we can afford to block all threads out of the map (and from manipulating its elements) for this operation.

```
1  void put(int index, ELEM_TYPE elem) {
2      pthread_rwlock_wrlock(&rwlock);//acquire rwlock write-mode
3      ElemWrapper *ew = (ElemWrapper*)malloc(sizeof(ElemWrapper));
4      pthread_mutex_init(&ew->mutex);
5      ew->elem = elem;
6      array[index] = ew;
7      pthread_rwlock_unlock(&rwlock);//unlock rwlock
8  }
9
10 ELEM_TYPE get(int index) {
11     pthread_rwlock_rdlock(&rwlock);//acquire rwlock in read-mode
12     ElemWrapper *ew = array[index];
13     pthread_mutex_lock(&ew->mutex);//lock item's mutex
14     return ew->elem;
15 }
16
17 void release(int index) {
18     ElemWrapper *ew = array[index];
19     pthread_mutex_unlock(&ew->mutex);//unlock item's mutex
20     pthread_rwlock_unlock(&rwlock);//release rwlock
21 }
22
23 ELEM_TYPE remove(int index) {
```

```
24      pthread_rwlock_wrlock(&rwlock);//acquire rwlock write-mode
25      // no other thread has valid ref to item
26      ElemWrapper *ew = array[index];
27      pthread_mutex_destroy(&ew->mutex);//destroy item's mutex
28      ELEM_TYPE el = ew->elem;
29      free(ew);
30      array[index] = NULL;
31      pthread_rwlock_unlock(&rwlock);//unlock rwlock
32      return el;
33  }
```

Listing 3.5: Simplified implementation of the map operations - The read-write lock solves the readers-writers problem and simplifies the removal of items by keeping count of the valid item references in the program.

3.2.7 Test suite

A test suite is available to ease the development and maintenance of `TCPSnitch`. All tests are high-level tests that observe the execution of `TCPSnitch` on a small program.[8] The tests are written in Ruby using the `MiniTest` testing framework which ships with the Ruby standard library. The test suite counts about 1600 lines of Ruby code, without blanks and comments. Tasks related to running the tests are automated with `rake`, a Ruby variant to the `make` program. When a test requires a peer to establish a TCP connection with, we start a local HTTP server using the `httpd` class from the Ruby standard library.

While the small programs executed by the tests are written in C, the C code itself is generated by a Ruby script. This allows to easily reuse and reassemble different blocks of code. In total, 111 test programs are generated. Each program targets a specific function intercepted by `TCPSnitch` or a specific scenario. The tests cases are grouped into three test suites:

`test_tcpsnitch.rb`: covers the shell wrapper script and the initialization module of the library. For instance, we assert that `TCPSnitch` behaves properly for each supported option.

`test_libc_overrides.rb`: tests each intercepted libc function to ensure that `TCPSnitch` does not crash the main program when intercepting the function. This module also asserts the presence of events in the JSON trace.

`test_tcp_events.rb`: tests the format of events in the JSON traces. We merely test the presence of keys and their data type but not their exact value.

[8] There are no unit tests that cover individual methods in the C code.

3.3 Architecture of `tcpsnitch.org`

The `tcpsnitch.org` platform is written with the Ruby on Rails framework. We wrote about 2100 lines of code, without blanks and comments, on top of the starting Rails application. Anyone familiar with the framework should be able to easily navigate through the code. This section only highlights some high-level architectural choices. The software stack is composed of the following main components:

- Puma acts as the Ruby application server.
- Nginx acts the web server in front of Puma.
- MongoDB stores trace events and quantitative analyses of traces.
- PostgreSQL stores the rest of the permanent data (described hereafter).
- Redis stores background jobs.
- Memcached serves for caching fragments of HTML pages.

Fig. 3.2 shows the main tables stored in the PostgreSQL database. Three types of entities are at the core of `tcpsnitch.org`:

App traces: An app trace represents a trace uploaded to `tcpsnitch.org` and mainly stores the metadata collected by `TCPSnitch`. An app trace contains one or many process traces.

Process traces: A process trace represents the trace of a single process and stores the name of the process and the log file of `TCPSnitch`. A process trace contains one or many socket traces.

Socket traces: A socket trace corresponds to the JSON trace of a single socket and primarily stores information about the socket itself.

This architecture is visible when browsing `TCPSnitch` as the user can visualize each trace at these three levels of granularity.

The events themselves, which compose the JSON traces, are stored in MongoDB and do not appear in Fig. 3.2. A single app trace may contain hundreds of thousands of events. This sheer volume of events is the main reason for using MongoDB. Because of its different data consistency model, MongoDB is more suited than a relational database such as PostgreSQL for working with that amount of data. MongoDB also offers the additional benefit of working with JSON documents. Since events are already in JSON format, we import them almost unmodified into MongoDB. Events are enriched with a few additional fields to keep track of the traces to which they belong and to speed up computations on the dataset. For instance, because the operating system is often used as a filtering criteria when querying the dataset, we store this information on each event.

Fig. 3.2 shows the `StatCategory` table, which contains the categories of

statistics computed by `tcpsnitch.org` on all traces and the dataset. These categories appear in the left menu of most pages. A `StatCategory` contains one or many `Stats`. A `Stat` encapsulates the logic for computing a metric on a given trace. For instance, a `Stat` contains a `filter` to select only a subset of the events and a `node` to indicate on with value of the event the metric should be computed. It also includes a `stat_type` which marks the kind of metric to compute (e.g. CDF, proportion or time series). The main reason for storing both `StatCategories` and `Stats` in PostgreSQL is to make it convenient for `TCPSnitch` administrators to add new metrics via the administrative interface. Via this interface, administrators can manage the traces and the metrics computed on these traces.

To be complete, the MongoDB also contains an `Analysis` collection. Each analysis document caches the metrics computed on a given trace or a segment of the dataset. Finally, the PostgreSQL database holds an `AdminUser` table, which is self-explanatory.

3.4 Conclusion

We have implemented `TCPSnitch`, an application that intercepts network library calls on the Linux and Android platforms to collect information about their usage, including the parameters passed to those API calls. A limitation is that applications can bypass `TCPSnitch` by either being statically linked with the C library or directly using the system calls. We also developed `tcpsnitch.org`, a platform designed to centralize, visualize and analyze the collected traces. Our hope is that this platform will enable the community to get a better understanding of networked application behaviors.

The `TCPSnitch` source code is available from `https://github.com/GregoryVds/tcpsnitch` with plenty of instructions about the installation procedure and the usage of `TCPSnitch`. The source code for `tcpsnitch.org` is also publicly available at `https://github.com/GregoryVds/tcpsnitch_web`.

After reading this chapter, the reader should be equipped to start contributing to and extending `TCPSnitch`. `TCPSnitch` currently supports Linux and Android. Further work remains to be done to support other platforms, starting with macOS.

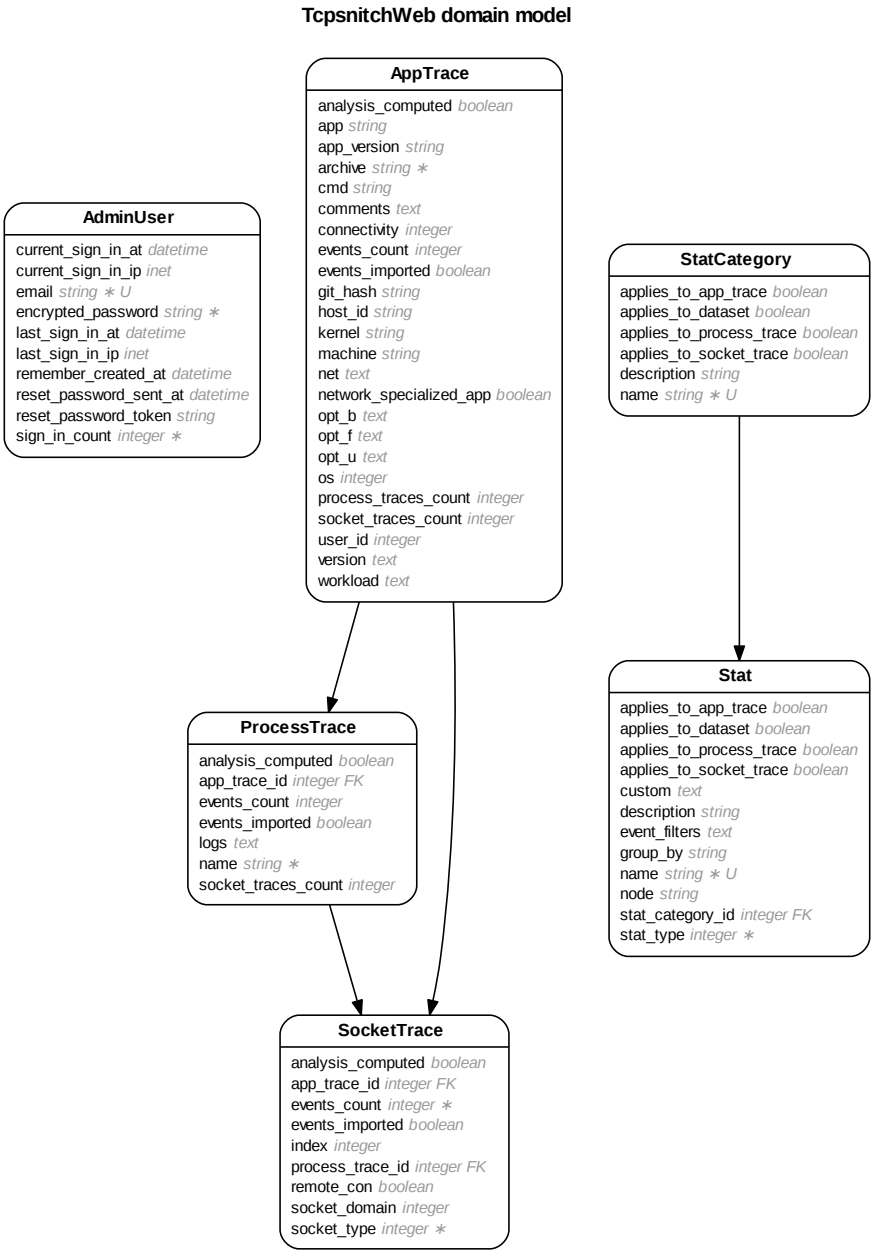

Figure 3.2: PostgreSQL domain model - This schema was automatically generated by the Rails ERD gem.

Chapter 4

Dataset

The previous chapter detailed the inner-workings of `TCPSnitch`. It explained the structure of the traces and discussed the information they contain. We also introduced `tcpsnitch.org`, a public database exposing our dataset. This chapter describes this dataset quantitatively.

Using `TCPSnitch`, we recorded traces for more than 130 different applications on Linux and Android and intercepted about 5 M function calls. We manually interacted with each application for a few seconds in order to reproduce a typical usage. In total, we tracked their interactions with 24 K Internet sockets. This chapter intends to give the reader an overview of the data available in the public dataset.

Section 4.1 describes the applications that compose the dataset. Section 4.2 analyzes the impact of frameworks and libraries on the API usage. Section 4.3 investigates the function calls present in the dataset. Finally, section 4.4 analyzes the nature of the sockets opened by the applications.

4.1 Application traces

Android

The Android dataset includes traces for 90 different applications. We mainly selected popular consumer-oriented applications from different categories of apps in the Google Play Store. The dataset currently does not include any server-side application. Table 4.1 shows a sample of representative applications. At the time of writing, we recorded all Android traces on Android 6.0.1 with an LG Nexus 5 device. The Android dataset includes 181 application traces that opened about 16 K sockets. This represents about 2.4 M intercepted function calls.

Category	Applications
Social	Facebook, Twitter, Linkedin
Streaming	Spotify, Netflix, Soundcloud
Video-telephony	Skype, Viber, Hangout
Shopping	Amazon, AliExpress, Zalando
Browsers	Chrome, Firefox, Opera
Productivity	Evernote, Slack, Mega
Video/photo	Youtube, Instagram, Pinterest

Table 4.1: Sample applications - The Android dataset contains traces of 90 applications from different categories of apps in the Google Play Store.

For some popular Android applications, we recorded multiple traces over both LTE and WiFi. We also recorded traces in perturbed network environments. For instance, we recorded WiFi traces with the access point configured with a bandwidth limit of 20 Kbps. For other traces, we started recording the trace near the access point and progressively walked away to force a WiFi to LTE handover. Our goal was to observe the patterns of interactions used by the applications that best cope with such situations. Looking at aggregate data, we did not notice any notable difference. The same functions, socket options, and `ioctl()` requests seem to be used in the different network environments. We also tried to manually inspect individual socket traces to detect interesting patterns of interactions but came up empty-handed. Further work probably remains to be done in that area.

It is important to note one caveat about the utilization of `TCPSnitch` on Android applications. On Android, applications usually do not call `exit()` because they typically remain running or idle once started. To end the tracing of an Android application, `TCPSnitch` calls the `force-stop` command of the activity manager tool (`am`) to terminate the application. This means that the application does not get the opportunity to cleanly `close()` its opened sockets. This caveat only affects the interception of the `close()` function.

Linux

The Linux dataset includes 45 different applications. In total, it includes 81 application traces for about 8 K opened Internet sockets. This represents another 2.6 M intercepted function calls. The applications composing the dataset fall in four main categories:

- Command-line applications used by computer scientists such as `curl`, `git`, `apt-get` or `rsync`.
- Specialized networking applications such `dig`, `traceroute` or `nmap`.

Language	Ruby	Python	JavaScript	Perl	PHP	Curl
Library	`Net:HTTP`	`urllib2`	`https mod`	`LWP::Simple`	`file_get_contents`	`libcurl`
API calls	186	353	106	465	920	420
Bytes recv	65 KB	175 KB	175 KB	178 KB	175 KB	175 KB
Main API call	`read()`	`read()`	`read()`	`read()`	`fcntl()`	`recv()`
Socket option	`TCP_NODELAY`	`SO_TYPE`	`SO_ERROR`	`SO_ERROR`	`SO_ERROR`	N/A
Async I/O	`ppoll()`	N/A	N/A	`select()`	`poll()`	`poll()`

Table 4.2: HTTP GET with different libraries - An HTTP GET request over TLS yields different API footprints with different libraries.

These applications often exhibit peculiar interactions with the socket API. Because they tend to skew the results, we exclude them when computing statistics on the Linux dataset.

- Traditional end-user applications with a graphical interface such as `Skype`, `Transmission`, `Firefox`, `Spotify` or `WeeChat`.
- Toy programs that are presented in the next section.

4.2 Impact of frameworks and libraries

In accordance with [5], we confirm that high-level frameworks and libraries drive the API usage. To demonstrate the impact of libraries, we wrote five Linux applications performing the same simple task in different programming languages. Each program executes an HTTP GET request over TLS for a 167 KB file. For each language, we searched on Google for a simple way to make an HTTP GET request and implemented the program according to the first pertinent search result. We also traced the execution of the Curl program on the same task. Table 4.2 contrasts the API footprint of these programs on selected dimensions.

Table 4.2 shows that the total number of API calls varies greatly, from 186 for the Ruby program to 920 for the PHP program. In reality, the Ruby program receives only 65 KB of data because the `Net:HTTP` library uses HTTP compression by default. While `read()` is the most often called API function for four programs, the main API call for Curl is `recv()` and the PHP programs calls `fcntl()` more frequently. In fact, the PHP program keeps toggling the socket `O_NONBLOCK` flag between `read()` calls for no apparent reason. This explains why the PHP program uses many more API calls than the other programs. Although all programs except Curl use a single socket option, this option differs. The JavaScript, Perl, and PHP programs use `SO_ERROR` after a non-blocking `connect()`. On the other hand, the Ruby and the Python programs perform a blocking `connect()` and do not use this option. Ruby uses the `TCP_NODELAY` option to disable Nagle's algorithm, and for some unknown reason, Python uses `SO_TYPE` to retrieve the socket type (`SOCK_STREAM`). Finally, Ruby, Perl, PHP, and Curl use the

socket in non-blocking mode during the data transfer but rely on three different functions for polling the socket: `ppoll()`, `select()` and `poll()`. Python and JavaScript use the socket in synchronous mode during the data transfer.

Considering that we displayed major differences in the API footprint of these toy programs, it comes as no surprise that we discovered major differences in the API usage patterns on Android and Linux. For instance, the most popular API functions differ and applications use different recurring combinations of socket options. The next sections highlight some of these differences.

4.3 Functions usage

The socket API defines many functions that often have overlapping purposes. For instance, there are as many as seven functions to send data: `write()`, `send()`, `sendto()`, `writev()`, `sendmsg()`, `sendmmsg()` and `sendfile()`. This section intends to shed some light about the real usage of the socket API functions.

Android

Some of our observations are dependent on Android 6.0.1. For instance, Bionic, Google's implementation of `libc`, implements some API functions by calling their more complex sibling, e.g. `send()` is implemented by calling `sendto()`. When the simpler version of these «twin» functions is called, `TCPSnitch` records two consecutive function calls although the application code only performs a single call. In addition to `send()` calling `sendto()`, we noticed that `recv()` calls `recvfrom()`, `accept()` calls `accept4()`, and `epoll_wait` calls `epoll_pwait()`. This means that the popularity of `sendto()`, `recvfrom()`, `accept4()`, and `epoll_pwait()` is overestimated in Fig. 4.1.

Fig. 4.1 shows the number of applications using each intercepted function on Android. It reveals that some functions are used by a surprisingly large fraction of the Android applications. For instance, all applications use `getsockopt()`, `setsockopt()`, and `fcntl()`. Only VLC does not call `getsockname()`. Another surprising result is that 86 of our client Android applications use `bind()`, a textbook server-side function. We observed that about 95% of these `bind()` calls specify `INADDR_ANY` as the IP address and 0 for the port number (meaning an OS assigned random port), but explicitly request for an IPv6 address. This usage is mainly driven by the `Socket` class

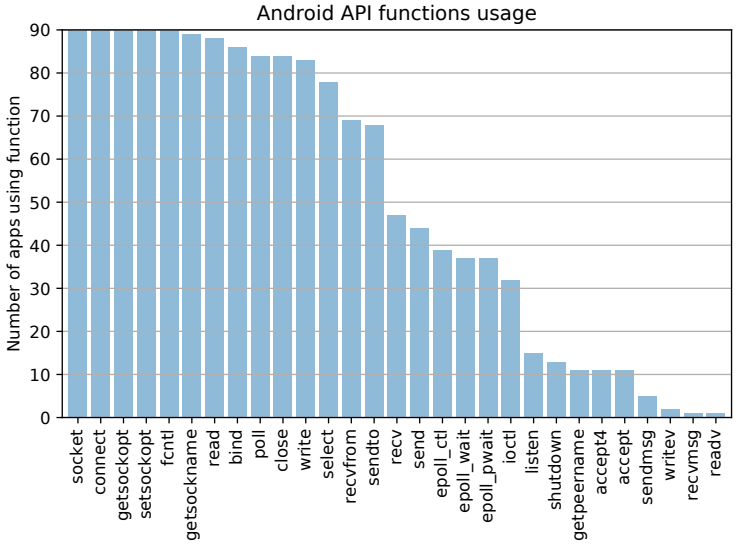

Figure 4.1: API functions usage on Android - A dozen API functions are used by almost all applications. Vectored I/O functions are mostly unused.

of the Android SDK [2] that caches the local address of the socket, using `getsockname()`, before trying to connect it.

Fig. 4.1 also shows that vectored I/O functions such as `writev()` and `sendmsg()` are seldom used an Android. These functions are optimizations that allow writing multiple noncontiguous blocks of data to a socket with a single system call.

Being used by respectively 84 and 78 applications, `poll()` and `select()` are the most widely used interfaces for operating asynchronous sockets. Only 39 applications use the Linux-specific `epoll` interface. As expected, most of these calls ask to be notified about incoming data available for reading.

Finally, we note that 11 functions that `TCPSnitch` intercepts are not shown in Fig. 4.1 because they were never used:

- The `sendfile()`, `sendmmsg()` and `recvmmsg()` functions are optimizations mostly useful for server-side applications requiring high-performance. For instance, `sendfile()` is a Linux-specific call that saves a back-and-forth copy between kernel and user space when sending a file over a socket, while `sendmmsg()` and `recvmmsg()` allow sending or receiving multiple `struct msghdr` in a single system call. As the Android dataset does not include any server-side application, the disuse of these functions is not surprising.

- The `pselect()` and `ppoll()` functions are also never used. These

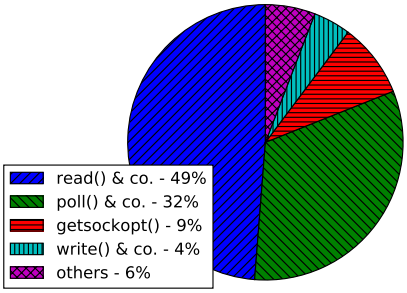

Figure 4.2: API functions calls on Android - Functions such as `read()` and `poll()` dominate the workload. Surprisingly, `getsockopt()` account for 9% of all function calls.

functions are stronger versions of their well-known siblings `select()` and `poll()` that protect against race conditions when using signals.

- The `sockatmark()` function is used for working with out-of-band data, which only has meaning for stream sockets. For instance, TCP calls it *urgent data*. The disuse of this function is coherent with the observation that the `MSG_OOB` sending flag, which allows sending out-of-band data, is never used on Android. In this context, the `sockatmark()` function becomes useless since its only purpose is to determine whether the read pointer is positioned on an out-of-band data mark.

- The `fdopen()` function wraps a file descriptor into a `FILE` pointer, which can be used with the buffered I/O standard library. As buffering may introduce undesirable latency on Internet sockets, the disuse of `fdopen()` is expected for client-side applications that strive to appear highly responsive to their users. For instance, we observed that 60% of the applications use the `TCP_NODELAY` socket option to disable Nagle's algorithm on TCP connections and decrease the network latency.

- The `dup` family of functions works on file descriptors, not only sockets. As the POSIX standard illustrates [15], a common usage of the `dup()` function is for I/O redirection, for instance, to redirect the standard output stream to a file. As Internet sockets do not really fit in this schema, it is not surprising to see them unused. Moreover, these functions are redundant with `fcntl()` that offers the same service.

While Fig. 4.1 tells us how widespread the usage of a given function is, Fig. 4.2 gives another perspective on the functions usage by showing how often functions are called. As expected for client-side applications, which are mostly data consumers, we observe that `read()` and its sibling functions dominate the function calls. After `read()`, come functions for working

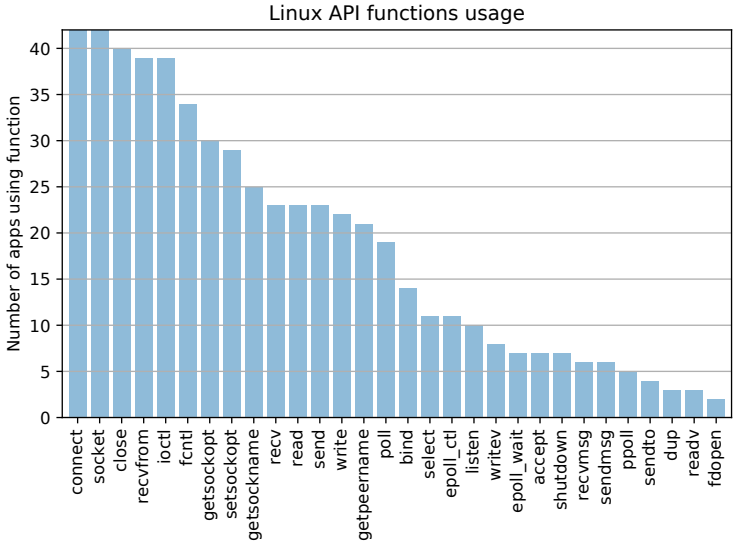

Figure 4.3: API functions usage on Linux - Linux applications appear more heterogeneous in their functions usage. Only five functions are used by more than 80% of the Linux applications.

with asynchronous sockets such as `poll()`. This is expected because these functions are often used together with `read()` functions. More surprisingly, `getsockopt()` accounts for 9% of all function calls. The next two chapters investigate this unexpected result.

Linux

Fig. 4.3 shows the number of applications using each intercepted function on Linux. The maximum is 42, the number of Linux applications, excluding `dig`, `traceroute` and `nmap`.

Compared to Android, we notice that Linux applications appear more heterogeneous in their functions usage. Whereas a dozen functions are used by almost all Android applications, only five functions are used by more than 80% of the Linux applications. While all Android applications must use the Android SDK, Linux applications have more freedom when it comes to choosing frameworks and libraries. In particular, we observe that only two functions are used by all Linux applications: `socket()` and `close()`. Because `TCPSnitch` is able to close applications cleanly on Linux, many more `close()` calls are intercepted compared to Android.

While the popularity of many functions is similar to Android, we discern some notable differences. Whereas `bind()` is used by most Android appli-

cations, only a third of the Linux applications use `bind()`. We also notice that most Linux applications use `ioctl()` while only a third of the Android applications use this interface.

Another striking difference appears when looking at the function calls (not shown in Fig 4.3): `getsockopt()` represents less than 0.01% of the calls on Linux. This stands in stark contrast with Android where `getsockopt()` accounts for a staggering 9% of the function calls.

4.4 Nature of sockets

Android

In the IPv6 enabled WiFi network used for the experiments, all but one application established a TCP connection with a remote host over IPv6. This confirms the growing importance of IPv6. All the surveyed applications opened at least one IPv6 socket while only 57 apps opened an IPv4 socket. Overall, 63% of all opened sockets were IPv6 sockets while 37% were IPv4 sockets.

Regarding the socket types, 73% of all opened sockets are `SOCK_STREAM` sockets and 27% are `SOCK_DGRAM` sockets. For 90% of the `socket()` calls, 0 was passed as the socket protocol. This tells the TCP/IP stack to use the only supported protocol given the socket type: UDP for `SOCK_DGRAM` and TCP for `SOCK_STREAM`. All our Android applications use TCP while only 31 applications use UDP sockets.

While all applications use asynchronous sockets, a single application used the `SOCK_NONBLOCK` optional flag when calling `socket()`. `SOCK_CLOEXEC` was never used. In fact, most sockets are made asynchronous after their creation using `fcntl(F_SETFL)` and five applications used `ioctl(FIONBIO)`. Usually, TCP sockets are turned asynchronous just before the `connect()` call. As a matter a fact, `O_NONBLOCK` is the only file status flag used by the studied Android applications. The `O_APPEND`, `O_ASYNC`, `O_DIRECT` and `O_NOATIME` flags were never used.

Linux

The Linux traces were mostly recorded in a virtual-machine environment without IPv6 connectivity to the Internet. All applications opened an IPv4 socket and 23 applications opened an IPv6 socket. We observed six applications using the happy eyeballs algorithm but failing to establish a TCP connection over IPv6: Firefox, Thunderbird, Lima, Transmission,

qBittorrent and TIN. Other applications opened TCP connections with local daemons over IPv6.

Although Android UDP sockets were explicitly requested by the applications, the situation is different on Linux. When applications use functions such as `getaddrinfo()`, the UDP sockets opened by libc are intercepted by `TCPSnitch`. As a result, all Linux applications open at least one UDP socket.

While `socket()` flags are unused on Android, 34% of the Linux `socket()` calls set the `SOCK_NONBLOCK` flag and 9% set the `SOCK_CLOEXEC` flag. Similarly to Android, `O_NONBLOCK` is the only file status flag used by the Linux applications with `fnctl(F_SETFL)` and `fnctl(F_GETFL)`.

4.5 Conclusion

Using `TCPSnitch`, we recorded traces for more than 130 different applications on Linux and Android, and observed major differences in the API usage patterns on both platforms. To confirm the observation of [5] that high-level frameworks and libraries drive the API usage, we contrasted the API footprint of six programs performing the same task using different libraries. Compared to Android, we noticed that Linux applications appear more heterogeneous in their functions usage. Due to these disparities, we argued that both datasets should be analyzed separately.

Further work remains to be done to expand the Linux dataset. At the moment, it is not really representative of the real-world usage because it mostly includes specialized command-line applications used by computer scientists. For this reason, the rest of this document will focus on the Android dataset. When the Linux dataset is expanded, contrasting in more detail the API usage on both platforms should yield interesting results.

This chapter showed that a large proportion of Android applications use functions such as `getsockopt()` and `bind()`. However, we have not yet provided any explanation for these surprising observations. The next two chapters bridge this gap by respectively dissecting the UDP and TCP sockets usage on Android.

Chapter 5

UDP sockets

The previous chapter gave an overview of the public dataset available on `tcpsnitch.org`. We mentioned that 31 Android applications use UDP sockets. In this chapter, we dissect how applications use these datagram sockets.

Overall, a total of 31 applications opened about 4 K `SOCK_DGRAM` sockets, which represents about 27% of all opened sockets. Note that those UDP sockets were explicitly requested by the applications themselves. UDP sockets opened for resolving domain names are not intercepted on Android. This chapter shows that UDP sockets are mainly used as a shortcut to retrieve information about the network configuration.

Section 5.1 analyzes the API functions usage on Android UDP sockets. Section 5.2 investigates for what purpose these sockets are being opened. Section 5.3 focuses on UDP sockets that send or receive data.

5.1 Functions usage

Fig. 5.1 shows the number of Android applications using each intercepted function on UDP sockets. The maximum is 31, the number of applications using UDP. We notice that among the 40 functions that `TCPSnitch` intercepts, only 20 are ever used with UDP sockets.

While `ioctl()` ranked among the least widely used functions for the global dataset, `ioctl()` surprisingly surges as the third most widely used function on UDP sockets. We observe the opposite trend for functions such as `setsockopt()`, `getsockopt()` or `getsockname()`. Although all applications use these functions when looking at the global dataset, less than half of the applications use them on UDP sockets. We also notice that a minority

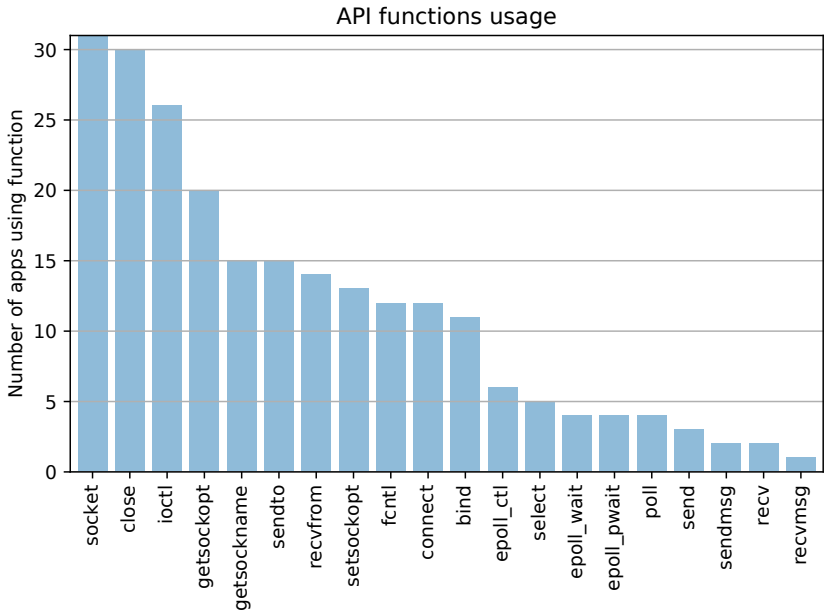

Figure 5.1: API functions usage on UDP sockets on Android - The third most widely used function is `ioctl()`. A minority of applications use `sendto()` and `recvfrom()`.

of UDP applications use `sendto()` and `recvfrom()`. This indicates that about half of the applications using UDP do not send or receive *any* data over UDP.[1]

5.2 Sockets usage

By looking at the amount of data sent/received over UDP sockets, we noticed that 85% of the opened `SOCK_DGRAM` sockets do not send or receive *any* data. These sockets only retrieve information about the networking environment using `ioctl()`. We also noticed that 8% of the `SOCK_DGRAM` sockets issue a `connect()` call but do not send or receive any data. In fact, only 6% of the observed `SOCK_DGRAM` send or receive data. Fig. 5.2 summarizes these findings.

Observing that a staggering 85% of UDP sockets only retrieve information about the network was surprising. However, this is not an isolated behavior. While the Hangouts application alone accounts for 30% of these sockets, 24 different applications account for the rest. Although those `ioctl()` requests apply to any socket, we suspect that applications

[1] The other functions `send()`, `recv()`, `sendmsg()`, and `recvmsg()`, are actually used by the same set of applications that uses `sendto()` and `recvfrom()`.

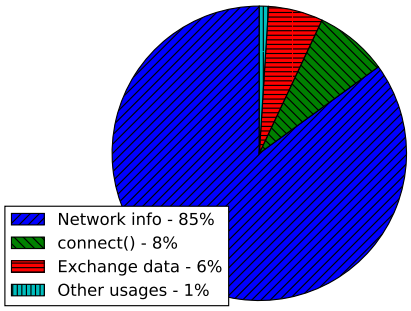

Figure 5.2: UDP sockets usage - Most UDP sockets do not send or receive *any* data but get information about the network environment using `ioctl()`.

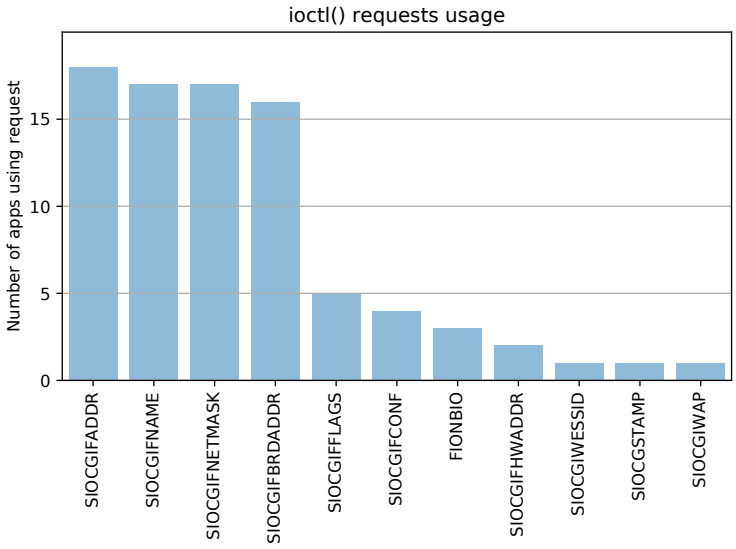

Figure 5.3: `ioctl()` requests usage for UDP sockets on Android - `SIOCGIFADDR`, `SIOCGIFNAME`, `SIOCGIFNETMASK` and `SIOCGIFBRDADDR` are used by a majority of applications on Android (max is 31).

use them on UDP sockets because they are cheaper to allocate than their TCP counterpart. On Linux and Android, our experiments revealed respectively a 20% and 13% speed-up to perform `open()` and `close()` on UDP sockets compared to TCP sockets.

Fig. 5.3 shows that four `ioctl()` requests are used by a majority of applications: `SIOCGIFADDR`, `SIOCGIFNAME`, `SIOCGIFNETMASK` and `SIOCGIFBRDADDR`. These requests allow retrieving respectively the network device address, name, network mask and broadcast address. We also noticed recurring combinations of `ioctl()` requests retrieved on the same socket. For instance, the `SIOCGIFADDR`, `SIOCGIFBRDADDR` and `SIOCGIFNETMASK` triplet was retrieved, in this order, on hundreds of sockets.

5.3 Sending and receiving data

Only 6% of the `SOCK_DGRAM` sockets sent or received data. Overall, 16 different applications opened these sockets: 5 are video-telephony apps such as Google Hangout or Skype, 4 are video or music streaming applications such as Spotify or Netflix and 3 are Google applications likely using QUIC such as Chrome or Google Plus. The 4 others are various applications that only exchange a few hundred bytes such as Shazam or Angry Birds.

Applications mainly use `sendto()` and `recvfrom()` to send or receive data. In contrast to TCP sockets, the `read()` and `write()` functions are unused with UDP sockets. `TCPSnitch` allows looking at the return values of these functions. Fig. 5.4 shows the cumulative distribution function of the number of bytes sent and received by a single function call. We observe that about 35% of the receiving calls return exactly 1 byte. These calls were made by three video-telephony apps: Hangouts, Skype and Messenger. A closer look reveals that 99% of these calls set the `MSG_PEEK` flag to determine if there is data in the receive queue. We also notice that 40% of the calls return exactly 1350 bytes. Coincidentally, this is the default packet size used by QUIC over IPv4. As expected, these packets were sent by Google applications: Youtube, Hangouts, Chrome and Google Plus. Apart from the 1-byte packets, sending calls tend to send smaller packets. About 50% of the sending calls send 50 bytes or fewer.

We also observed that 29% of the receiving calls set the `MSG_PEEK` to peek on the receiving queue without removing data and that 0.6% of the sending calls set the `MSG_NOSIGNAL` flag to prevent a `SIGPIPE` signal from being raised in the case of error. We did not find any indication of the usage of the other flags on `SOCK_DGRAM` sockets. We noticed that Messenger uses the `SIOCGSTAMP` iotcl during video calls roughly every second `recvfrom()`. This ioctl allows round trip time measurements.

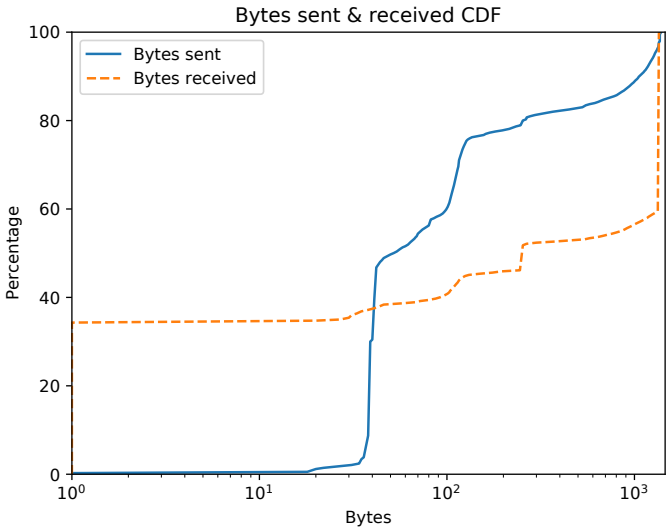

Figure 5.4: CDF of the number of bytes sent/received by a single call on Android UDP sockets - 35% of the receiving calls return exactly 1 byte and are performed by video-telephony apps. 40% of the receiving calls return exactly 1350 bytes, the default packet size used by QUIC over IPv4.

Multicast is one of the use cases for UDP sockets. We observed eight applications using UDP sockets to send multicast packets but only two applications joined multicast groups. These two applications use multicast to discover other similar applications on the same LAN, e.g. using the Simple Service Discovery Protocol. A typical example is streaming applications that allow discovering another device where audio/video can be streamed over the network.

5.4 Conclusion

This chapter revealed that only 6% of Android UDP sockets send or receive data. We explained this surprising result by showing that UDP sockets are mainly used as a shortcut to retrieve information about the network configuration.

We showed that sending and receiving flags are mostly unused with UDP sockets, with the exception of `MSG_PEEK`. We observed that 35% of the `recv()` calls use a 1-byte buffer in conjunction with `MSG_PEEK` to determine if there is data in the receive queue. These calls were made by three video-telephony applications: Hangouts, Skype and Messenger. We also noticed that 40% of the received packets were exactly 1350 bytes long,

the default packet size for QUIC over IPv4.

Further work remains to be done to identify the higher-level frameworks and libraries that drive the API usage on UDP sockets. For instance, the hundreds of sockets that retrieve the same triplet of `ioctl()` requests mentioned in section 5.2 are probably operated by some function exposed by the Android SDK. In the next chapter, we turn our attention to TCP sockets.

Chapter 6

TCP sockets

Without any surprise, chapter 4 highlighted that all our Android applications use TCP. Overall, applications opened 12 K `SOCK_STREAM` sockets, which account for 73% of all opened sockets. In this chapter, we dissect the socket API usage on Android TCP sockets.

We saw in chapter 4 that all applications in our dataset use functions such as `getsockopt()` and `fcntl()`. However, we observed in the previous chapter that their usage drops when the dataset is restricted to UDP sockets. These functions must often be used on TCP sockets. This chapter uncovers recurring patterns of interactions with the TCP/IP stack. We explain why some functions are used by all applications by tracking down the higher-level frameworks and libraries responsible for these patterns.

Section 6.1 discusses sockets that do not call `connect()` or connect to a loopback address (local sockets). Section 6.2 discusses sockets that connect to distant servers (remote sockets). Section 6.3 analyzes the usage of socket options with TCP sockets.

6.1 Local sockets

Table 6.1 summarizes how applications use TCP sockets. We notice that 37% are local sockets that do not call `connect()` or interact with local daemons or applications. A staggering 27% of TCP sockets only call `setsockopt(SO_RCVTIMEO)` once or several times after the initial `socket()` call. Since `SO_RCVTIMEO` only modifies the receiving timeout of the target socket, these operations seem wasteful but we could not find a valid explanation for this behavior. Another 6% of TCP sockets only call `close()` after `socket()` and 3% call `ioctl(SIOCGIWNAME)` to determine if the current interface is WiFi or cellular before closing the socket.

37%	**Local sockets**
27%	`setsockopt(SO_RCVTIMEO)`
6%	Immediate `close()`
3%	Determine if interface is wireless
1%	Other usages
63%	**Remote sockets**
59%	Exchange data after `connect()`
4%	Do not send/recv data from network

Table 6.1: TCP sockets usage - 37% do not `connect()` or `connect()` to a loopback address while 63% `connect()` to a distant server. Most local sockets only call `setsockopt(SO_RCVTIMEO)`.

While 85% of the UDP sockets use `ioctl()` to retrieve information about the network, only 2.5% of TCP sockets call `ioctl()`. This supports our observation that applications prefer to execute `ioctl()` requests on UDP sockets because they are less costly to allocate.

6.2 Remotely connected sockets

We now restrict our analysis to the 7.5 K sockets that opened a TCP connection to a remote host. Various API calls could be used to create those connections. However, our analysis reveals a common pattern of 16 socket API calls to open such a connection. Fig. 6.1 illustrates this pattern. It results from the interactions between the IO part of the Java Android core library [2] and the `okhttp` external library [29]. We first notice a synchronous setup phase that binds the socket. The `setsockopt(SO_RCVTIMEO)` call is issued by the OkHTTP library. Then, the socket is put in non-blocking mode before the `connect()` call. The successive `fcntl()` calls flip the `O_NONBLOCK` bit while keeping the values of the other flags constant. The `getsockopt (SO_ERROR)` call checks whether the `connect()` succeeded. Then, the socket is turned synchronous again and we observe two redundant calls to `getsockopt(SO_RCVTIMEO)`, probably related to the TLS library [9]. Finally, the socket is put in non-blocking mode again before the TLS handshake. The two `getsockname()` calls are issued by the Android Java `Socket` class to cache the local address before and after the `connect()` call. This opening pattern explains why nearly all applications use functions such as `setsockopt()`, `getsockopt()`, `bind()` and `fcntl()`.

Surprisingly, 15 of our client applications use the `listen()` call. Among those, only 11 ever accepted an incoming connection with `accept()` or `accept4()`. Among the 449 incoming connections observed, 98% originated from a loopback address. We noticed five connections originating from a

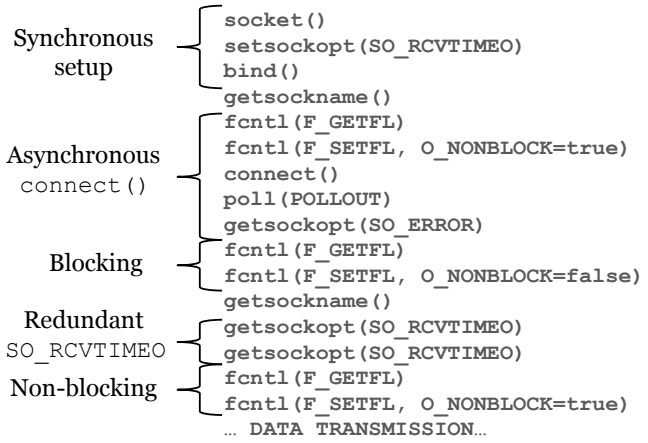

Figure 6.1: Opening pattern on TCP sockets - After a synchronous setup phase that binds the socket, a non-blocking `connect()` call is issued. After two redundant `getsockopt(SO_RCVTIMEO)`, the socket is turned in non-blocking mode again before transmitting data.

link-local address while three connections originated from a remote network. These three remote incoming connections were accepted by a single application, Skype, that uses NAT traversal.

Let us now focus our analysis on the data transfer. After the `connect()` call, 94% of the TCP sockets exchange data. Almost all applications use the generic `read()` and `write()` interfaces and only half of them use their dedicated socket counterparts, `recv()` and `send()`. Given the cost of issuing system calls, networking textbooks recommend using large buffers when transferring data. `TCPSnitch` allows dissecting how applications use each call. Fig. 6.2 shows a cumulative distribution function for the size of the buffer given to the various receive functions. Surprisingly, we observe that respectively 34% and 16% of the `recv()` and `recvfrom()` calls use a buffer of exactly 1 byte. Similarly to UDP sockets, more than 75% of these calls set the `MSG_PEEK` flag to determine if there is data in the receive queue. We also notice many calls with buffers of 3 and 5 bytes. The Lively application issued 99% of the 3 bytes calls, without any receiving flag set. Using small receiving buffers is however not an isolated behavior. Buffers of exactly 5 bytes were used by 84 different applications. Overall, about half of the `recv()` calls are given a buffer of 5 bytes or fewer. To be complete, we notice that powers of two like 4096 and 8192 are also often used for the buffer size.

Most functions to send and receive data support optional flags. The most popular sending flag, `MSG_NOSIGNAL`, is set for 60% of the calls. This flag requests not to send the `SIGPIPE` signal, which by default terminates the process, when an application writes to a disconnected socket. It is particularly useful for libraries since this flag does not modify the process

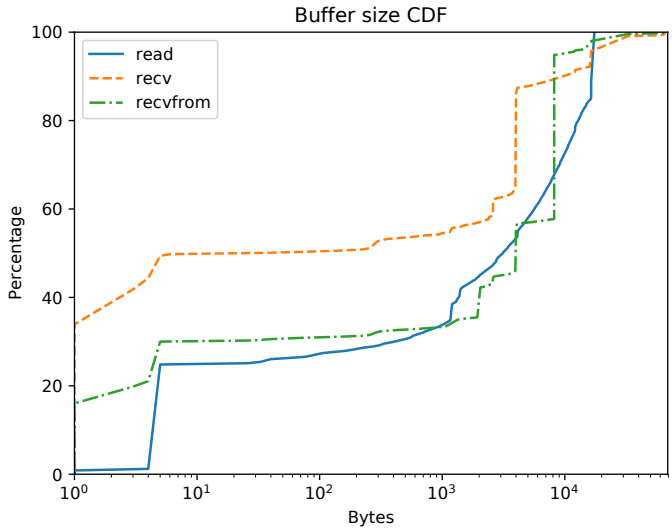

Figure 6.2: CDF of the buffer size passed to receive functions - While networking textbooks recommend using large buffers when transferring data, half of the `recv()` calls use a buffer of 5 bytes or fewer.

signal handlers. Only two other sending flags are used: `MSG_DONTWAIT` and `MSG_MORE`. The `MSG_DONTWAIT` flag is used to make 13% of the calls non-blocking and `MSG_MORE` is set on 2% of the calls to indicate that more data is coming. The other sending flags[1] are never used. The `MSG_DONTWAIT` flag is used to make 18% of the receiving calls non-blocking and 16% of the calls set the `MSG_PEEK` flag to peek on the TCP receive queue without removing data. Finally, a tiny fraction of those receiving calls (0.04%) set `MSG_WAITALL` to request the operating system to block until it has enough data to fill the buffer. The remaining flags[2] do not appear in our traces.

As observed for the connection establishment, there is a very frequent pattern for the termination of a connection. We noticed that 78 applications and about half of all opened sockets use `getsockopt(SO_DEBUG)` and `getsockopt(SO_LINGER)` before issuing `close()`. The utilization of `SO_DEBUG` at this point of the connection is surprising. We investigated the Android source code and confirmed its usage in the IO Java core library of Android [1] where a function closes all file descriptors. Because sockets using `SO_LINGER` need some additional processing to avoid the socket API `close()` call to block, a `getsockopt()` is issued to detect if the file descriptor is a socket. If this call succeeds, then the file descriptor is indeed a socket. It seems that a failed `getsockopt(SO_DEBUG)` is less critical from

[1] `MSG_CONFIRM`, `MSG_DONTROUTE`, `MSG_EOR` and `MSG_OOB`.
[2] `MSG_CMSG_CLOEXEC`, `MSG_ERRQUEUE`, `MSG_OOB`, `MSG_TRUNC`.

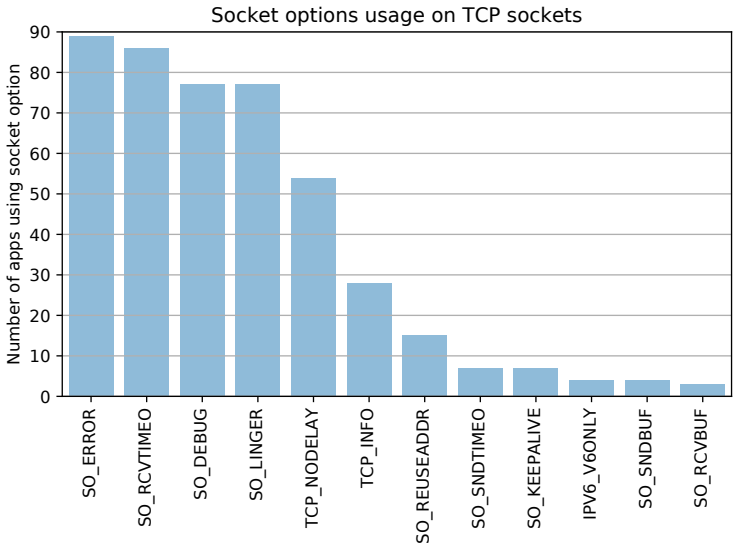

Figure 6.3: Number of applications using each socket option - `SO_ERROR` is often used after a non-blocking `connect()`. `SO_RCVTIMEO` appears at the beginning of most TCP connections. `SO_DEBUG` and `SO_LINGER` are used together before `close()`. `TCP_INFO` is used by a surprisingly large number of applications.

a performance viewpoint than a failed `getsockopt(SO_LINGER)`, hence its use. This closing pattern would certainly be observed for a much higher proportion of the sockets if `TCPSnitch` could terminate cleanly the traced Android applications.

6.3 Socket options

Applications can use socket options to tune the behavior of the underlying TCP/IP stack. Linux supports a growing number of non-standard socket options. Fig. 6.3 shows how many applications use the main socket options observed in our dataset. Several of these options are expected and some were discussed earlier. At connection establishment, `SO_ERROR` is used after a non-blocking `connect()` and `SO_RCVTIMEO` is both set and retrieved. `SO_DEBUG` and `SO_LINGER` are used together before `close()`. `TCP_NODELAY` disables Nagle's algorithm on TCP connections to decrease the network latency. This is expected from client applications that strive to appear highly responsive. Fig. 6.4 gives another perspective on socket options usage by showing how often they are used. We notice that `SO_RCVTIMEO` dominates all other options. In section 6.1, we explained that 27% of all TCP sockets only call `setsockopt(SO_RCVTIMEO)` after the initial `socket()` call. Fig. 6.4 also shows that `TCP_INFO` is the second most often used socket option. This

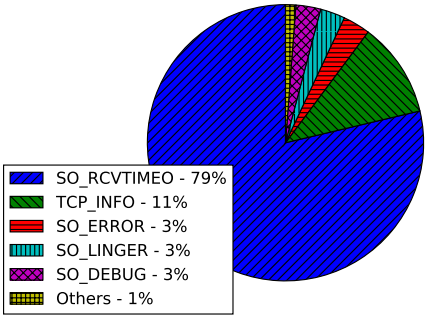

Figure 6.4: `getsockopt()` **and** `setsockopt()` **arguments for TCP sockets (local and remote)** - `SO_RCVTIMEO` is by far the most used argument. The non-standard `TCP_INFO` option is often retrieved. `SO_ERROR` is often used after a non-blocking `connect()`. `SO_LINGER` and `SO_DEBUG` are often used together before a `close()` call.

is more surprising.

`TCP_INFO` is a non-standard Linux TCP option that exports to the application counters maintained by the TCP stack. The standard Android Java API does not expose this socket option and applications must resort to a C/C++ library to use it. Still, 28 applications make use of this socket option. As expected, those are mostly highly popular applications such as Youtube, Chrome, Facebook or Spotify. For these applications, `TCP_INFO` was retrieved on 26% of the `SOCK_STREAM` sockets and 73% of these sockets retrieve `TCP_INFO` only once. Facebook, Messenger, and Instagram are the only applications that issue dozens of `TCP_INFO` on a single TCP connection. For instance, we observed a Facebook TCP connection lasting 32 seconds for which `TCP_INFO` was retrieved about 3 K times. These `TCP_INFO` calls do not specifically happen at the start or the end of a connection but seem uniformly distributed during the lifetime of the TCP connection. In fact, Fig. 6.5 shows that when `TCP_INFO` is retrieved multiple times on a given socket, the number of `TCP_INFO` calls seems to be a linear function of the number of `recv()` calls.

6.4 Conclusion

This chapter revealed that 27% of Android TCP sockets only perform `setsockopt(SO_RCVTIMEO)` calls after the initial `connect()`. Further work remains to be done to find a valid explanation for this behavior which seems wasteful. Whereas only 6% of UDP sockets exchange data with the network, we observed that 59% of TCP sockets exchange data. We also discovered that half of the `recv()` calls use buffers of 5 bytes or fewer.

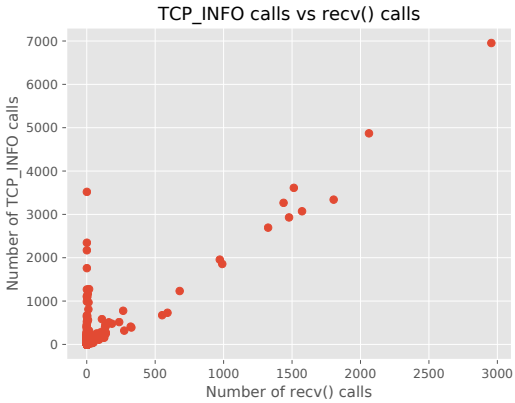

Figure 6.5: Number of `TCP_INFO` and `recv()` calls per socket - `TCP_INFO` is usually retrieved only once on a given socket, regardless of the number of `recv()` calls. For Facebook applications, the number of `TCP_INFO` calls seems to be a linear function of the number of `recv()` calls.

We uncovered recurring patterns of API calls at the opening and closing of a TCP connection. We identified the libraries responsible for these behaviors. These findings explained the unexpectedly widespread usage of functions such `bind()` and socket options such as `SO_DEBUG`.

We also discussed the unanticipated widespread usage of the `TCP_INFO` socket option. This non-standard Linux TCP option is the second most often used option with TCP sockets on Android and is usually retrieved only once on a given socket. For Facebook applications, the number of `TCP_INFO` calls seems to be a linear function of the number of `recv()` calls.

Chapter 7

Discussion

We have proposed and implemented `TCPSnitch`, an application that intercepts network library calls on the Linux and Android platforms to collect information about their usage, including the parameters passed to those API calls.

In addition to serving research purposes, we believe that `TCPSnitch` has educational value and could be used in classes that teach network programming. For instance, students could be asked to trace a simple application and explain what they observe in the traces. This would be a playful way for students to discover the different interfaces and options offered by the API. Students could also be challenged to identify programming best practices and pitfalls that appear in the traces. Another insightful exercise would be to contrast the API footprint of multiple applications like we performed in chapter 4.

Our application is open-source[1] and we encourage anyone interested in this project to contribute patches. We have many extensions in mind, among which:

- Supporting other platforms, starting with macOS.
- Improving the test suite, especially for Android.
- Supporting the `syscall()` function.
- Supporting the buffered I/O functions from the standard library.
- Improving the verbose mode, which provides a service similar to `ltrace` but dedicated to network calls.
- Capturing backtraces for library calls to help identify their origin.

Using `TCPSnitch`, we collected about 5 M API calls made by more than 130 applications on 24 K sockets. The collected dataset is publicly avail-

[1] Available on `https://github.com/GregoryVds/tcpsnitch`

able and can be explored on `tcpsnitch.org`. Visitors can browse the traces and visualize statistics about the entire dataset or a particular trace. We encourage people to contribute new traces to expand the dataset. In particular, the Linux dataset should be expanded and we would like to add traces of server applications. Like `TCPSnitch`, the code of `tcpsnitch.org` is also open-source[2]. Again, we encourage the community to contribute patches. Some directions for future work are the following:

- Allowing users to specify custom criteria for filtering the dataset and obtain fine-grained statistics about the dataset.
- Improving existing visualizations of traces (e.g. the plot showing the sum of bytes sent per socket over time is not very readable).
- Add new visualizations of the traces (e.g. the number of opened sockets over the trace lifetime).

Our analysis of the dataset confirmed the conclusion of [5] that high-level frameworks and libraries drive the socket API usage. We demonstrated that programs performing the same task using different libraries display different API footprints. We also highlighted major differences in the API usage patterns on Android and Linux. For instance, we demonstrated that the most popular API functions differ and that applications use different recurring combinations of socket options. Compared to Android, we noticed that Linux applications appear more heterogeneous in their functions usage.

In an IPv6 enabled WiFi network, we discovered that Android applications prefer IPv6 over IPv4 (on Android 6.0.1). In a virtual-machine environment without IPv6 connectivity to the Internet, we observed six Linux applications using the happy eyeballs algorithm but failing to establish a TCP connection over IPv6.

On Android, 11 functions intercepted by `TCPSnitch` are unused by the applications in our dataset. We detected some functions such as `getsockopt()` and `bind()` that are used by an unexpectedly large proportion of the applications. We made the same observation about socket options such as `SO_DEBUG` and `SO_ERROR`. Most applications use various socket options even if the Java API does not expose them directly. We uncovered recurring patterns of API calls that explain the widespread usage of these API functions and socket options. We also identified the libraries responsible for these interactions.

We showed that only 6% of UDP sockets send or receive data on Android. UDP sockets are mainly used as a shortcut to retrieve information about the network configuration using `ioctl()`. For UDP sockets exchanging data, we revealed that:

[2] Available on `https://github.com/GregoryVds/tcpsnitch_web`

- Sending and receiving flags are mostly unused, with the exception of `MSG_PEEK`.
- About 35% of the `recv()` calls use a 1-byte buffer in conjunction with the `MSG_PEEK` flag to determine if there is data in the receive queue.
- Another 40% of the received packets are exactly 1350 bytes long, the default packet size for QUIC over IPv4.

We observed that 27% of Android TCP sockets only perform `setsockopt(SO_RCVTIMEO)` calls after the initial `connect()`. While this behavior seems wasteful, we did not find a valid explanation for this observation. Among the 59% of TCP sockets that exchange data, we discovered that half of the `recv()` calls use buffers of 5 bytes or fewer. We also revealed that `TCP_INFO`, a non-standard Linux TCP option, is the second most often used option with TCP sockets on Android and is usually retrieved only once on a given socket. For Facebook applications, the number of `TCP_INFO` calls seems to be a linear function of the number of `recv()` calls.

`TCPSnitch` and its associated website already provide a good overview of how real applications use the socket API. In summary, we observed that the socket API usage in the wild differs greatly from textbook usage. Many Internet sockets do not exchange data, many API calls are redundant and many calls use tiny data buffers. This work opens the door for further work in several directions:

- Analyzing in greater depth the traces collected in a perturbed network environment. In particular, it would be interesting to observe the patterns used by the applications that best cope with such situations.
- Applying data mining algorithms to the dataset. While our analysis was mostly driven by intuition and manual inspection of the traces, we are convinced that data mining techniques could uncover new patterns of interactions, clusters of applications and socket traces, etc.
- Analyzing the Linux dataset and contrasting the API usage on Linux with Android. It would also be interesting to contrast the API usage of applications written in different languages and assess the impact of the libraries used by these languages.

Bibliography

[1] AOSP 6.0.1. Blockguardos. https://android.googlesource.com/platform/libcore/+/android-6.0.1_r79/luni/src/main/ java/libcore/io/BlockGuardOs.java#81.

[2] AOSP 6.0.1. Socket. https://android.googlesource.com/platform/libcore/+/android-6.0.1_r79/luni/src/main/java/ java/net/-Socket.java#223.

[3] Hasan Abbasi, Christian Poellabauer, Karsten Schwan, Gregory Losik, and Richard West. A quality-of-service enhanced socket api in gnu/linux. In *Proceedings of the 4th Real-Time Linux Workshop, Boston, Massachusetts*. Citeseer, 2002.

[4] Werner Almesberger, Leena Chandran-Wadia, Silvia Giordano, Jean-Yves Le Boudec, and Rolf Schmid. Using quality of service can be simple: Arequipa with renegotiable atm connections. *Computer Networks and ISDN Systems*, 30(24):2327–2336, 1998.

[5] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. Posix abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 19:1–19:17, New York, NY, USA, 2016. ACM.

[6] David M. Beazley, Brian D. Ward, and Ian R. Cooke. The inside story on shared libraries and dynamic loading. *Computing in Science and Engg.*, 3(5):90–97, September 2001.

[7] Package Cloud. How does ltrace work?, 2016. https://blog.packagecloud.io/eng/2016/03/14/how-does-ltrace-work.

[8] Package Cloud. How does strace work?, 2016. https://blog.packagecloud.io/eng/2016/02/29/how-does-strace-work.

[9] conscrypt. Opensslsocketimpl. https://android.googlesource.com/platform/external/conscrypt/+/android-6.0.1_r79/src/ main/java/org/conscrypt/ OpenSSLSocketImpl.java#259.

[10] Michael J Donahoo and Kenneth L Calvert. *The pocket guide to TCP/IP sockets: C version*. Morgan Kaufmann, 2001.

[11] Jinliang Fan, Jun Xu, Mostafa H. Ammar, and Sue B. Moon. Prefix-preserving ip address anonymization: Measurement-based security evaluation and a new cryptography-based scheme. *Comput. Netw.*, 46(2):253–272, October 2004.

[12] Center for Applied Internet Data Analysis. Summary of anonymization best practice techniquesup specification, 2016. https://www.caida.org/projects/predict/anonymization.

[13] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Linux Symposium*, volume 1, 2004.

[14] Brendan Gregg. Choosing a linux tracer, 2015. http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html.

[15] The Open Group. dup specification, 2016. http://pubs.opengroup.org/onlinepubs/009695399/functions/dup.html.

[16] B Hesmans, G Detal, S Barre, R Bauduin, and O Bonaventure. Smapp: Towards smart multipath tcp-enabled applications. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 28. ACM, 2015.

[17] Benjamin Hesmans and Olivier Bonaventure. An enhanced socket api for multipath tcp. In *Proceedings of the 2016 Applied Networking Research Workshop*, pages 1–6. ACM, 2016.

[18] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010.

[19] Samu; Tarkoma Sasu; Gurtov Andrei Komu, Miika; Varjonen. Sockets and beyond: Assessing the source code of network applications. D4, 2011.

[20] linux.org. ptrace man page, 2016. http://man7.org/linux/man-pages/man2/ptrace.2.html.

[21] Preethi Natarajan, Fred Baker, Paul D Amer, and Jonathan T Leighton. Sctp: What, why, and how. *IEEE Internet Computing*, 13(5), 2009.

[22] john k. ousterhout, hervé da costa, david harrison, john a. kunze, mike kupfer, and james g. thompson. A trace-driven analysis of the unix 4.2 bsd file system. *sigops oper. syst. rev.*, 19(5):15–24, December 1985.

[23] Niels Provos and Chuck Lever. Scalable Network I/O in Linux. In *USENIX 2000 Technical Conference, Freenix Track*, San Diego, CA, June 2000.

[24] John S. Quarterman, Abraham Silberschatz, and James L. Peterson. 4.2bsd and 4.3bsd as examples of the unix system. *ACM Comput. Surv.*, 17(4):379–418, December 1985.

[25] Stephen A Rago. *UNIX System V network programming*. Addison-Wesley Professional, 1993.

[26] Dennis M Ritchie. The unix system: A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[27] Michael Schier and Michael Welzl. Using dccp: Issues and improvements. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–9. IEEE, 2012.

[28] Philipp S. Schmidt, Theresa Enghardt, Ramin Khalili, and Anja Feldmann. Socket intents: Leveraging application awareness for multi-access connectivity. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 295–300, New York, NY, USA, 2013. ACM.

[29] SquareUp. Okhttp. https://android.googlesource.com/platform/external/okhttp/+/android-6.0.1_r79/okhttp/src/main/java/com/squareup/okhttp/internal/http/SocketConnector.java#147.

[30] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming, Vol. 1*. Pearson Education, 3 edition, 2003.

[31] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment, Second Edition*. Addison-Wesley Professional, 2nd edition, 2008.

[32] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern linux api usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 16:1–16:16, New York, NY, USA, 2016. ACM.

Appendix A

Appendix

A.1 List of intercepted functions

- accept()
- accept4()
- bind()
- close()
- connect()
- dup()
- dup2()
- dup3()
- epoll_ctl()
- epoll_pwait()
- epoll_wait()
- fcntl()
- fdopen()
- getpeername()
- getsockname()
- getsockopt()
- ioctl()
- isfdtype()
- listen()
- poll()
- ppoll()
- pselect()
- read()
- readv()
- recv()
- recvfrom()
- recvmmsg()
- recvmsg()
- select()
- send()
- sendfile()
- sendmmsg()
- sendmsg()
- sendto()
- setsockopt()
- shutdown()
- sockatmark()
- socket()
- write()
- writev()

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**