TCPSnitch: Dissecting the Usage of the Socket API

Gregory Vander Schueren[1], Quentin De Coninck[2] and Olivier Bonaventure[2]
Universite Catholique de Louvain, Louvain-la-Neuve, Belgium
[1]gregory.vanderschueren@gmail.be, [2]{first.last}@uclouvain.be

ABSTRACT

Networked applications interact with the TCP/IP stack through the socket API. Over the years, various extensions have been added to this popular API. In this paper, we propose and implement the `TCPSnitch` software that tracks the interactions between Linux and Android applications and the TCP/IP stack. We collect a dataset containing the interactions produced by more than 120 different applications. Our analysis reveals that applications use a variety of API calls. On Android, many applications use various socket options even if the Java API does not expose them directly. `TCPSnitch` and the associated dataset are publicly available.

1. INTRODUCTION

The socket API was introduced together with release 4.2 of the BSD Unix distribution that included a functional TCP/IP stack [14]. This API allows applications to interact with the underlying networking stack. 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 heart of the socket API is a set of basic system calls including `socket`, `bind`, `connect`, `accept`, `close`, `listen`, `send`, `receive`,... 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 [16] was extended to support the TCP/IP protocol stack and used in Unix System V [15].

Over the years, the popularity of the socket API grew in parallel with the deployment of the global Internet. Nowadays, the socket API is considered by many as the standard API for simple networked applications. Several popular textbooks are entirely devoted to this API [20, 7]. 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 example, 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 [13, 8] focused on improving the interactions between applications and the underlying stack through `poll/select`, `epoll`,... 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 [17, 12]. Socket extensions have also been proposed to deal with multihoming [18] and specific APIs have been implemented on top of Multipath TCP [9, 10].

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.

We propose `TCPSnitch`, an open-source software that collects detailed traces of the interactions between networked applications and the Linux TCP/IP stack and sends them to a publicly available database exposing various statistics. This paper is organized as follows. We first describe `TCPSnitch` in section 2. Then we present in section 3 the traces that we collected from 90 different Android applications. Section 4 analyses in more details the utilization of UDP sockets by those applications while section 5 focuses on the TCP sockets. We summarize the main findings of this work and future work in section 6.

2. TCPSNITCH

Different solutions have been proposed and implemented to analyze the utilization of system and library calls by applications. Two approaches are possible. The first one is to analyze the application code (binary or sometimes source for open-source applications)

and extract the interesting calls from the corresponding files. Several researchers have adopted this approach to study networked applications. In 2011, [11] analyzed the source code of 2187 Ubuntu applications to detect the presence of certain keywords of the socket API. In 2016, [21] disassembled binaries of 30K Linux applications using `objdump` and performed a call-graph analysis to study the Linux API usage. Still in 2016, [5] proposed `libtrack` and analyzed 1.1M Android applications for linkage with POSIX functions. The main advantage of this approach is that it is possible to analyze a large number of applications to determine the system calls used by the majority of the applications. Unfortunately, it is very difficult to determine which parameters are passed to these identified functions or how frequently they are called. Source code analysis is also impractical for closed-source applications.

The second approach is to instrument the application and intercept the system or library calls. On Unix variants, the `strace` or `ltrace` applications can be used to collect traces of the system or library calls. The `libtrack` tool proposed by [5] also supports dynamic tracing of functions invocations. `TCPSnitch` currently intercepts 40 functions that are related to the network stack. `TCPSnitch` tracks the functions that are applied on each socket with their timestamp, parameters and return value. It also collects metadata information such as system information, the network configuration and the kernel version. Compared to simpler tools like `strace` and `libtrack`, a major benefit of `TCPSnitch` is that all the data collected during the utilization of an application can be uploaded on a public database. The web interface of this database, available on `https://tcpsnitch.org`, provides different visualizations of the database and allow users to browse through the collected traces. `TCPSnitch` is written in C and counts about 6500 lines of code, without blanks and comments.

Compared to the first approach, the main advantage of `TCPSnitch` is that it can trace sequences of calls and also collect information about the function parameters and the return values. This enables us 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,... While static analysis tools such as [11] or [21] give indications about the possible usage of some socket API calls or options, `TCPSnitch` allows observing their actual use. Since `TCPSnitch` uses `LD_PRELOAD` like `strace` to intercept the functions calling the system calls in the standard C library, it is possible for applications to bypass `TCPSnitch` by either being statically linked with the C library or directly using the system calls. Before analyzing the results, it is important to note one caveat about the utilization of `TCPSnitch` on Android applications. On Android, applications do not usually 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, not other functions.

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 data. Furthermore, all the non-loopback or link-local IP addresses that are collected as parameters of the traced system 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.

3. DATASET

Using `TCPSnitch`, we recorded traces for 90 Android and 33 Linux applications by manually interacting with each application for a few seconds in order to reproduce a typical usage. We mainly selected popular consumer-oriented client applications and the dataset currently does not include any server-side application. For some popular applications, we recorded multiple traces in different network environments.

We observed 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. In accordance with [5], we confirm that high-level frameworks and libraries drive the API usage and are at the root of such glaring disparities. Due to space limitations, we restrict our analysis to the Android dataset for the rest of this paper. The full dataset with various visualizations is publicly available from `https://tcpsnitch.org`.

Our Android dataset mostly includes highly popular applications from different categories of apps in the Google Play Store. Table 1 shows a sample of representative applications. At the time of writing, all Android traces have been recorded on Android 6.0.1 with a LG Nexus 5 device. In total, the Android dataset includes 181 application traces that opened a total of 16.384 sockets. This represents about 2.3M intercepted function calls.

3.1 Usage of the socket API functions

The socket API contains various functions that often have overlapping purposes. For instance, there are as many as 7 functions to send data: `write()`, `send()`, `sendto()`, `writev()`, `sendmsg()`,`sendmmsg()` and `sendfile()`. Figure 1 shows the number of applications

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

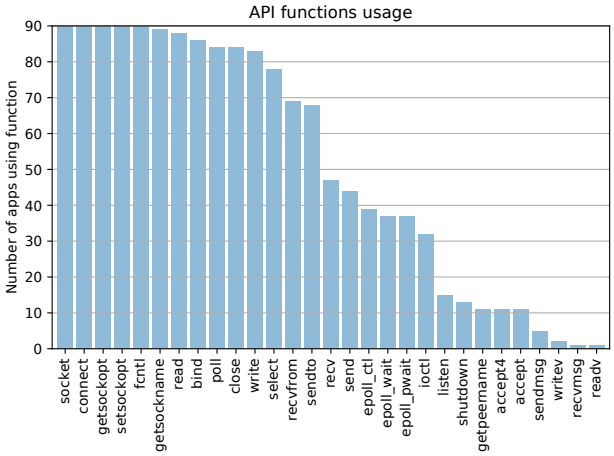

Figure 1: API functions usage. A dozen API functions are used by almost all applications. Vectored I/O functions are mostly unused.

using each intercepted function. This section intends to shed some light about the real usage of these functions.

Some functions are used by a large fraction of the applications. For instance, `getsockopt()`, `setsockopt()` and `fnctl()` are used by all the applications in our dataset and only one application does not call `getsockname()`. Another surprising result is that a textbook server-side function such as `bind()` is used by 96% of our client Android applications. We observe 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 of the Android SDK [2] that caches the local address of the socket (using `getsockname()`) before trying to connect it.

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 simple version of these 'twin' functions [1] is called, `TCPSnitch` records 2 consecutive function calls although the application code actually performs a single function call. This means that the popularity of `sendto()`, `recvfrom()`, `accept4()` and `epoll_pwait()` is overestimated on Fig. 1.

We did not observe any utilization of `sendfile()`, `sendmmsg()` and `recvmmsg()`. These 3 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 to send or receive multiple `struct msghdr` in a single system call. Figure 1 also shows that vectored I/O functions such as `readv()` and `recvmsg()` are seldom used an Android.

[1]Here is a complete list of these twin functions: `send()` calls `sendto()`, `recv()` calls `recvfrom()`, `accept()` calls `accept4()` and `epoll_wait` calls `epoll_pwait()`.

85%	Get info about network with `ioctl()`
8%	`connect()` but does not exchange data
6%	Send or receive data
1%	Other usages

Table 2: UDP sockets usage. Most UDP sockets do not send or receive *any* data but get information about the network environment using ioctl().

3.2 Types of sockets

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 is a confirmation of the growing importance of IPv6. All the surveyed applications opened at least one IPv6 socket while only 64% opened an IPv4 socket.

While all applications use asynchronous sockets, a single application used the `SOCK_NONBLOCK` optional flag when calling `socket()`. `SOCK_CLOEXEC` was never used. Most sockets are made asynchronous after their creation using `fcntl(F_SETFL)` and 5 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 with `fnctl(F_SETFL)` and `fnctl(F_GETFL)`. The `O_APPEND`, `O_ASYNC`, `O_DIRECT` and `O_NOATIME` flags were never used.

4. UDP SOCKETS

We first analyze how UDP is used by the 31 applications in our dataset that open at least one `SOCK_DGRAM` socket. Note that those UDP sockets are explicitly requested by the applications themselves since `TCPSnitch` does not track `getaddrinfo()` and related functions that are part of `libc`.

Looking at the amount of data sent/received over those UDP sockets, we noticed that 85% of the opened

	Request	Purpose (get dev *)
44%	SIOCGIFADDR	Address
25%	SIOCGIFNAME	Name
20%	SIOCGIFFLAGS	Active flag word
5%	SIOCGIFNETMASK	Network mask
5%	SIOCGIFBRDADDR	Broadcast address
1%	Others	N/A

Table 3: ioctl() requests breakdown. 85% of the UDP sockets use these requests to get information about the network devices.

SOCK_DGRAM sockets do not send or receive *any* data. Those sockets are created to retrieve information about the networking environment using `ioctl()`. While a single application opened 30% of these sockets, 15 applications use UDP and never send UDP data. Table 3 details the main `ioctl()` requests. Although those `ioctl()` apply to any socket, we suspect that applications perform them on UDP sockets because they are cheaper than their TCP counterpart.

Overall, 16 applications sent or received data over UDP: 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 like Chrome or Google Plus. The rest 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. We 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` from being raised in 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. Among the `SOCK_DGRAM` sockets that we observed, only 6% sent or received data. It is interesting to note that 8% of the `SOCK_DGRAM` sockets issued a `connect()` without sending or receiving any data.

Multicast is one of the use cases for UDP sockets. We observed 8 applications that used UDP sockets to send multicast packets but only 2 applications joined multicast groups. These 2 applications use multicast to discover other similar applications on the same LAN, e.g. using the Simple Service Discovery Protocol. A typical example are streaming applications that allow to discover another device where audio/video can be streamed over the network.

5. TCP SOCKETS

Without much surprise, all our Android applications use TCP. `SOCK_STREAM` sockets account for 73% of all

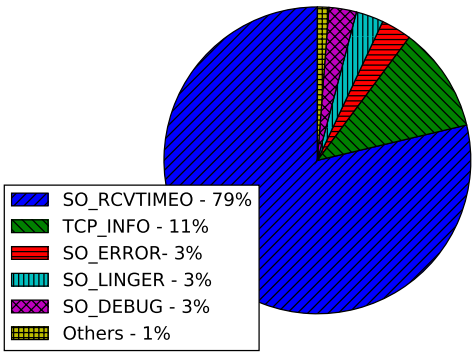

Figure 2: getsockopt() and setsockopt() arguments for all TCP sockets (local and remote). SO_RCVTIMEO is by far the most used argument. SO_ERROR is often used after a non-blocking connect(). The non-standard TCP_INFO option is often retrieved. SO_LINGER and SO_DEBUG are often used together before a close() call.

opened sockets. 63% of these TCP sockets `connect()` to a remote address while 37% do not call `connect()` or `connect()` to a loopback address. We first briefly analyze these later sockets that interact with local daemons or applications. We then analyze in more details the sockets that connect to distant servers.

5.1 Local sockets

We observe that a staggering 73% of the local sockets only call `setsockopt(SO_RCVTIMEO)` once or several times after the initial `socket()` call. As a result, figure 2 shows how the `SO_RCVTIMEO` socket option dominates the `setsockopt()` and `getsockopt()` arguments for TCP sockets (both local and remote). 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 16% of the local sockets only call `close()` after `socket()` and 3% call `ioctl(SIOCGIWNAME)` to determine if the current interface is wireless before closing the socket. Table 4 summarizes these findings. While 85% of the UDP sockets use `ioctl()` to retrieve information about the network, we rarely observe `ioctl()` on TCP sockets. This supports our observation that applications prefer to perform `ioctl()` requests on UDP sockets because they are less costly.

5.2 Remotely connected sockets

We now restrict our analysis to the 7505 TCP connections that were used to contact a remote host. Various system 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. This pattern is illustrated in figure 3. It results from the interactions between the IO part of the Java Android core

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 4: TCP sockets usage. 37% do not connect() or connect() to a loopback address while 63% connect() to a remote address. Most local sockets only call setsockopt(SO_RCVTIMEO).

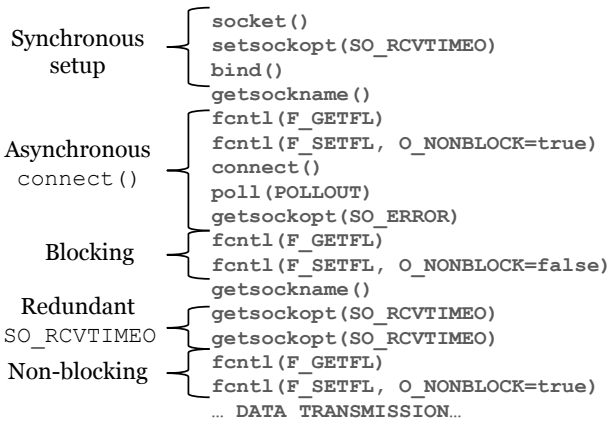

Figure 3: Opening pattern on TCP sockets. After a synchronous setup phase that binds the socket, a non-blocking connect() call is issued. After 2 redundant getsockopt(SO_RCVTIMEO), the socket is turned in non-blocking mode again before the transmission of data.

library [2] and the `okhttp` external library [19]. We first observe 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 modify the `O_NONBLOCK` bit while keeping the values of the other flags. 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 [6]. 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` to cache the local address before and after the `connect()` call.

Surprisingly, 15 applications use the `listen()` call. Among those, only 11 ever accepted an incoming connection with `accept()` or `accept4()`. Among the 449 incoming connection observed, 98% originated from a loopback address. We noticed 5 connections originating from a link-local address while 3 connections originated from a remote network. These 3 remote incoming connections were accepted by a single application, Skype, that uses NAT traversal.

Let us now focus our analysis on the data transfer. 94% of the TCP sockets exchange data after the `connect()` call. Almost all applications use the generic `read()` and `write()` calls. Only half of them use their dedicated socket counterparts, `recv()` and `send()`. Given the cost of issuing system calls, networking textbooks recommend to use large buffers when transferring data. `TCPSnitch` allows to dissect how applications use each call. Figure 4 shows a cumulative distribution function for the size of the buffer given the 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 and we also observe a lot of 5 bytes long buffers. Overall, about half of the `recv()` calls are passed a buffer of 5 bytes or less.

These functions support optional flags. The most popular sending flag is `MSG_NOSIGNAL` which is set on 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 signal handlers. Only two other sending flags are used: `MSG_DONTWAIT` and `MSG_MORE`. 13% of the calls are non-blocking thanks to the `MSG_DONTWAIT` flag. `MSG_MORE` is set on 2% of the calls to indicate that more data is coming. The other sending flags[2] are never used. 18% of the receiving calls are turned non-blocking using the `MSG_DONTWAIT` flag 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[3] do not appear in our traces.

As observed for the connection establishment, there is a very frequent pattern for the termination of a connection. 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 a performance viewpoint than a failed `getsockopt(SO_LINGER)`, hence its use. This closing pattern would certainly be observed on a much higher

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

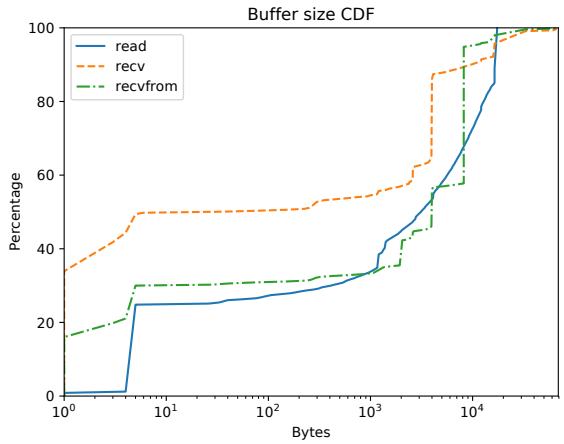

Figure 4: Cumulative distribution function of the buffer size passed to the different receive functions. Half of the recv() calls use a buffer of 5 bytes or less.

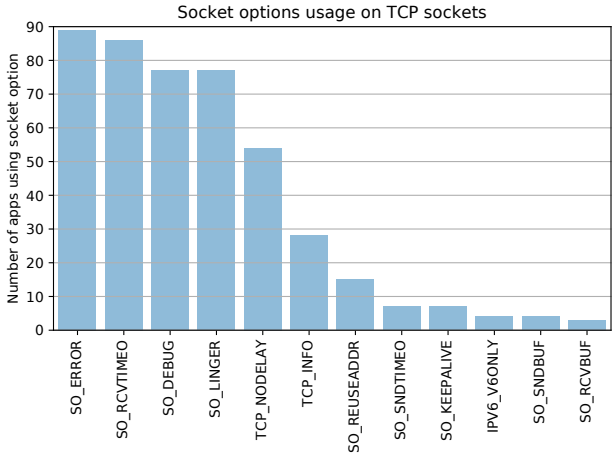

Figure 5: Number of applications using each socket option. SO_ERROR is often used after a non blocking connect(). SO_RCVTIMEO appears at the beginning of most TCP connections. SO_DEBUG and SO_LINGER are used together before close(). TCP_INFO is used by a surprisingly large number of applications.

proportion of the sockets if `TCPSnitch` could terminate cleanly the traced Android applications.

5.3 Socket options

Socket options can be used by applications to tune the behavior of the underlying TCP/IP stack. Linux supports a growing number of non-standard socket options. Figure 5 shows how many applications use the main socket options observed in our dataset. Several of these options are expected and some were discussed earlier, `TCP_INFO` was more surprising. This non-standard Linux TCP option 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 by 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 where `TCP_INFO` was retrieved about 3000 times, almost as often as the 3500 `recv()` calls on the same connection. 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. As figure 2 shows, `TCP_INFO` is the second most used socket option argument for TCP sockets.

6. DISCUSSION

We have proposed `TCPSnitch`, an application that intercepts network system and library calls on the Linux and Android platforms to collect more information about their usage, including the parameters passed to those API calls. We collected more than 2.3 millions calls made by 90 popular applications on sixteen thousands sockets. The application and the collected dataset are publicly available[4].

Our analysis revealed several interesting patterns for the utilization of the socket API on Android applications. First, in an IPv6 enabled WiFi network, these applications prefer IPv6 over IPv4. Second, UDP sockets are mainly used as a shortcut to retrieve information about the network configuration. Third, many Android applications use the same pattern of system calls to establish and terminate TCP connections. Fourth, Android applications use various socket options, even some like `TCP_INFO` that are not directly exposed by the standard Java API.

`TCPSnitch` and its associated website already provide a good overview of how real applications use the socket API. Our future work will be to add traces from more applications in the database and support other platforms starting with MacOS.

[4]The entire dataset can be explored via `https://tcpsnitch.org`. The `TCPSnitch` source code is available from `https://github.com/GregoryVds/tcpsnitch` and the web interface can be retrieved from `https://github.com/GregoryVds/tcpsnitch_web`.

7. REFERENCES

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

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

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

[9] 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.

[10] 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.

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

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

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

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.