Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

Analysing repacked Telegram and Signal with use of observability and security tools

Diploma thesis

2024 Ing. Bc. Jakub Škoda

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

Analysing repacked Telegram and Signal with use of observability and security tools

Diploma thesis

Study programme:	Computer Science
Field of study:	Computer Science
Department:	Department of Computer Science
Supervisor:	doc. RNDr. Daniel Olejár, PhD.
Consultant:	Mgr. Peter Košinár

Bratislava 2024 Ing. Bc. Jakub Škoda

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Ing. Jakub Škoda
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Analysing repacked Telegram and Signal with use of observability and security tools

Annotation: The aim of the diploma thesis is to analyze repacked version of Signal and Telegram instant
messaging applications using various observability and security tools. Main purpose of this
thesis is first to efficiently recognize if an application uses either Telegram or Signal in the
background, to verify if it performs the same function that legitimate original application
does and to highlight any non-standard and possibly malicious behaviour.

Aim: The aim of the diploma thesis is to analyze repacked version of Signal and Telegram instant
messaging applications using various observability and security tools.

Keywords: repackaging attacks, malware, observability tools, security tools, black-box analysis, behavioral analysis, Signal, Telegram, Linux, Wireshark, strace

Supervisor:	doc. RNDr. Daniel Olejár, PhD.
Consultant:	Mgr. Peter Košinár
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	08.01.2024

Approved: 10.01.2024 prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

I would like to thank my supervisor doc. RNDr. Daniel Olejár, PhD. for helping me with the choice of topic and all the academic advice.
I also want to thank my consultant Mgr. Peter Košinár for always finding time for me, productive discussions and for his friendly approach.

Abstract

ŠKODA, Jakub: Analysing repacked Telegram and Signal with use of observability and security tools. [Diploma thesis] – Comenius University Bratislava. Faculty of Mathematics, Physics and Informatics; Department of Computer Science. – Supervisor: doc. RNDr. Daniel Olejár, PhD., Consultant: Mgr. Peter Košinár, Bratislava: FMPH UNIBA, 2024, 67 p.

In this diploma thesis, we analyse repacked versions of Signal and Telegram instant messaging applications in the Linux environment with the use of various observability and security tools. In Chapter 1, we introduce past and current examples of repackaging attacks and define the functioning of the attack. A repackaging attack is a frequently used malware distribution method for various operating systems, especially for the Android operating system. In the Linux environment, there is not enough up-to-date research on this type of attack. In Chapter 2, we focus on selected observability tools and security tools in the Linux ecosystem and clarify the difference between them. Chapter 3 provides an overview of selected forensic analyses of Telegram, Signal, and other instant messaging applications for various operating systems. We highlight the most important findings and what has changed since their publication. Finally, in Chapter 4 we present the output of multiple Linux tracing tools used to compare the official signal-desktop application with the unofficial signal-cli, Telegram, as well as minimal examples of Qt and Electron frameworks.

Keywords: behavioural analysis, black-box analysis, eBPF, Linux, malware, observability tools, repackaging attacks, security tools, Signal, strace, Telegram, Wireshark

Abstrakt

ŠKODA, Jakub: Analýza prebaleného Telegramu a Signalu pomocou pozorovacích a bezpečnostných nástrojov. [Diplomová práca] – Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra informatiky. – Vedúci diplomovej práce: doc. RNDr. Daniel Olejár, PhD., Konzultant: Mgr. Peter Košinár, Bratislava: FMPH UNIBA, 2024, 67 s.

V tejto diplomovej práci analyzujeme prebalené verzie aplikácií na okamžité zasielanie správ Signal a Telegram v prostredí operačného systému Linux pomocou rôznych pozorovacích a bezpečnostných nástrojov. V kapitole 1 predstavujeme minulé a súčasné príklady prebaľovacích útokov a definujeme fungovanie útoku. Prebaľovací útok je často používaný spôsob distribúcie škodlivého softvéru pre rôzne operačné systémy, najmä pre operačný systém Android. V prostredí operačného systému Linux nie je dostatok aktuálneho výskumu o tomto type útokov. V kapitole 2 stručne popisujeme vybrané pozorovacie a bezpečnostné nástroje v ekosystéme Linux a objasňujeme rozdiel medzi nimi. Kapitola 3 obsahuje prehľad vybraných forenzných analýz aplikácií Telegram, Signal a ďalších aplikácií na okamžité zasielanie správ pre rôzne operačné systémy. Zdôrazňujeme najdôležitejšie zistenia a to, čo sa od ich uverejnenia zmenilo. V kapitole 4 uvádzame výstupy viacerých pozorovacích nástrojov pre operačný systém Linux, ktoré sme použili na porovnanie oficiálnej aplikácie signal-desktop s neoficiálnou signal-cli, Telegramom, ako aj s minimálnymi príkladmi frameworkov Qt a Electron.

Kľúčové slová: analýza čiernej skrinky, analýza správania, bezpečnostné nástroje, eBPF, Linux, malware, pozorovacie nástroje, prebaľovací útok, Signal, strace, Telegram, Wireshark

License

Contents

List of Figures

List of Tables

Introduction

This thesis is concerned with analysing how instant messaging applications interact with their environment (system calls, internet packets), while treating the applications themselves as a black-box.

There are two approaches of investigating and detecting malicious behaviour of software applications.

The first approach is based on the analysis of the source code of an application. The source code of the studied application can be obtained by reverse engineering. The code is analysed to see if it includes any malware-like patterns, for example file operations such as mass encryption, deletion, sending or altering of files that the given application has no reason to access, running code, or downloading payloads that seem unrelated to the claimed purpose of the software.

We will use another approach, based on observation of the application behaviour. The application is treated as a black-box and the analysis using various observability tools (further discussed in Chapter 2) tries to find suspicious activities.

This observation approach is often used for detection and analysis of repackaging attacks. The principle of a repackaging attack (described in more detail later) is following: the attacker takes a legitimate application, inserts malicious code into it, and then offers the application to the public. Sometimes the attacker poses as the original vendor of the application, or the application is offered as an improved version of the original software. We explore recent instances of these attacks in Chapter 1.

We apply the observation approach to analyse various versions of Signal and Telegram instant messaging applications. Signal is well regarded for its security and open-source approach – everything from text through calls to gifs that you send is end-to-end encrypted by default. Even the amount of metadata visible to Signal's servers is limited to a bare minimum. Telegram is mostly open-source as well, offers optional end-to-end encrypted chat, and it has been a frequent target of recent repackaging attacks.

To analyse various versions of Signal and Telegram applications, we will create a standard pattern of behaviour for legitimate versions of both Telegram and Signal. We will analyse and document the system calls they use, the internet packets they send and receive as well as any other relevant distinct interaction that the observability tools we use will be able to notice.

The repackaging attacks are best known from the Android operating system, where most of the researchers' attention is focused. For this reason, we decided to focus on the Linux operating system, where there is less research.

The reader should be familiar with the Linux operating system, the basics of malware terminology and networking.

1 Existing repackaging attacks

A repackaging attack is the use of repacked versions of legitimate applications with malicious payloads. The first part of the attack consists of obtaining a legitimate application, disassembling it, enclosing additional (often malicious) payloads, and then re-assembling it. For open-source and source-available software, or in cases where the source code is leaked, this situation is even easier as the whole reverse engineering part can be skipped. In the second part of the attack, targeted users must be tricked or enticed into downloading and installing modified (malicious) applications instead of the legitimate version. [1], [2]

Not all of these repacked applications are malicious. Many of them are a form of application plagiarism, where paid or simply popular applications are repacked to include paid advertisements and in-app purchases. Such plagiarised applications are then distributed by the repackager to generate revenue. [2], [3] Targeted application owners lose on average 14% of their advertising revenue to application plagiarism assuming that users who downloaded the plagiarized versions would have used the original applications instead. However, sometimes repackaging might be even completely legitimate such as rebranded applications (applications from the same developer with a high degree of code similarity) [3] or just white-label applications or a fork of existing open-source projects.

1.1 Rise of repackaging attacks on Android

Repackaging attacks were popular during the rise of the Android mobile operating system. In 2012, [1] collected 1,260 malware samples from various Android markets – stores that distribute and sell applications for the Android operating system. Of these 1,083 (or 86.0%) were repacked versions of legitimate applications with malicious payloads.

In 2013, [3] crawled 265,359 free applications from 17 Android markets around the world (73.7% of the applications are from the official Google Play market, 14.7% from 9 third-party English markets, 13.8% from 6 third-party Chinese markets, and 0.46% from 2 Russian markets). Out of the sampled applications, 44,268 (16.7%) were cloned applications. The authors define a cloned application as “an application that is a modified copy of another

application and thus shares a significant portion of its application code with the original", which is rather interchangeable with our term "repacked application".

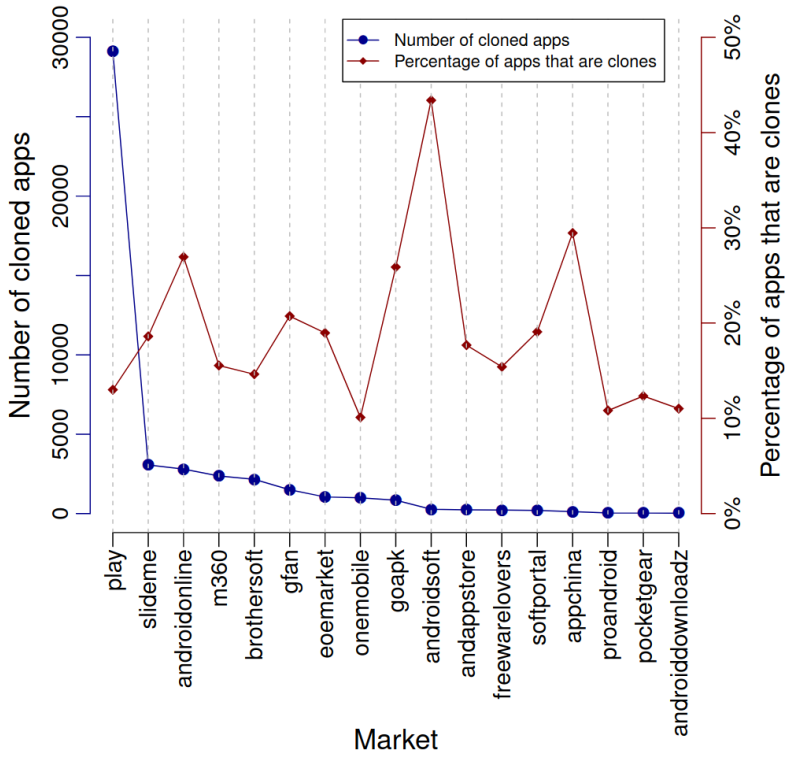

Figure 1: Applications sampled in [3] by market, ©Clint Gibler et al., 2013

Figure (1) shows the abundance of cloned applications by market. The absolute number of cloned applications from each market is represented by the axis labelled "Number of cloned apps". The axis labelled "Percentage of apps that are clones" is simply the result of dividing the number of cloned applications from a particular market by the total number of applications from that market in the entire database.

Most of the sampled applications came from Google Play, around 195,570 out of 265,359. It is not surprising that in absolute terms the greater number of cloned applications was identified there, nearly 30,000 cloned applications, which is just over 10% of the sampled Google Play applications. In relative terms, however, Google Play had one of the lowest percentages of cloned applications. Outside of Google Play, [3] sampled around 69,790 applications. Out of these samples, AndroidSoft had the most cloned applications, with almost 50% of the sampled applications from this store being clones. This was followed

by the Chinese markets AndroidOnline, GoApk and AppChina, each of which had almost 30% of cloned applications in their respective samples.

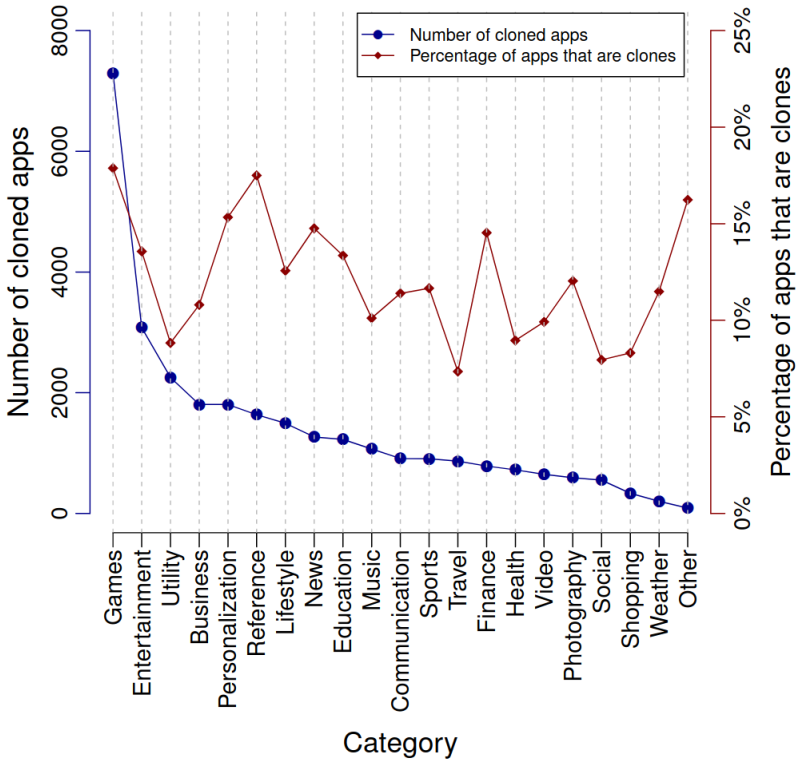

Figure 2: Applications sampled in [3] by category, ©Clint Gibler et al., 2013

Figure (2) is fairly similar to the previous one, just displays applications by category, i.e. what they are used for, instead of by market. The "Number of cloned apps" axis represents the absolute number of cloned applications in each category. The "Percentage of apps that are clones" is the result of the number of cloned applications for a specific category divided by the total number of applications from that category in the entire database.

Games is the most frequently cloned category, both in absolute terms, with over 7,000 applications, and in relative terms, with around 18% of games in the database being clones. Another popular category is *Reference*, which includes Books & Reference, E-books, Ebooks & Reference, Reference, with around 17% of cloned applications. Popular is also a vague category *Other* with around 16% of clones, which includes Developer/Programmer, Home & Hobby, Other, Religion, then *Personalization*, which includes Personalization and

Wallpapers, with over 15% of cloned applications. The remaining categories have between 7% and 15% of clones, e.g. the category *Communication* has around 11% of clones.

1.2 Recent repackaging attacks

Over the past five years, the trend of malicious repacked applications on Android has continued. As security measures on Google Play have been tightened, it is now common for attackers to trick users into downloading malicious applications from sources outside the application markets, such as websites that mimic the application's legitimate website or Telegram channels. However, there are still numerous examples of malicious repacked applications on Google Play and other stores. Frequently, their aim is to steal cryptocurrency funds, or they are simply part of a more targeted spear-head attack.

In 2023 there was a large group of repacked Telegram and WhatsApp applications targeting mainly Chinese users as both applications are banned in China. Threat actors set up Google Ads that led to fraudulent YouTube channels (reported by the authors of [4] and now taken down by Google) which included links to copycat Telegram and WhatsApp websites. [4]

Maliciously repacked versions of both Telegram and WhatsApp targeted cryptocurrency funds using malware called clippers. According to the authors, Lukas Štefanko and Peter Strýček [4], this is the first instance of clippers built into instant messaging applications. In addition, some of the maliciously modified Telegram applications used optical character recognition (OCR) to read text in screenshots and photos to steal a seed phrase used for recovering cryptocurrency wallets – this is the first instance of Android malware using OCR. Other attacks monitored Telegram for keywords related to cryptocurrencies and sent the full message to the attacker if the keyword was recognised. In some cases, they even exfiltrated internal Telegram data and basic device information. Beside the Android operating system, the attackers also targeted Windows, where the malicious Telegram used remote access trojans (RATs) that enabled full control of the victim's system. [4]

The clippers mentioned above are a type of malware that steals or modifies the contents of the clipboard. It is well-suited for stealing cryptocurrency because addresses of online

cryptocurrency wallets are composed of long strings of characters, which are mostly copied and pasted using the clipboard rather than typed manually. Malware simply switches the victim's cryptocurrency wallet address for the attacker's address in chat communication, with the addresses either being hardcoded or dynamically retrieved from the attacker's server. [4] Android developers have tried to respond to these clipper attacks and prevent their occurrence. For Android version 10 and higher clipboard data can only be accessed by the default input method editor (IME) or the application that currently has the focus. [5] Despite these new security measures, clipper attacks are still possible.

Many malicious applications are often promoted via various Telegram channels. This is also the case of the WhatsApp mod, which offers users additional features. It was not malicious from the start, but later became infected by a trojan called Trojan-Spy.AndroidOS.CanesSpy. It was active from mid-August 2023 until at least around October 2023. It was advertised on Telegram channels, where the most popular channel had almost two million subscribers. The Telegram channels were mostly in Arabic and Azeri languages, and the affected regions were mainly Azerbaijan, Saudi Arabia, Yemen, Turkey, and Egypt. [6]

In some cases, a repacked application can pose as a completely different service. This is the case of trojanised Telegram, active in November 2021 that included the StrongPity backdoor code and posed as an official application of the random-video-chat service Shagle. Shagle is a real service that provides encrypted communication between strangers, however, the Shagle service is only accessible through their official website and Shagle does not provide an Android application. The malicious application was distributed via a copycat website, mimicking the Shagle service. The StrongPity backdoor is modular, all necessary binary modules are encrypted using AES and downloaded from a C&C server. [7]

Google Play is a still viable channel for repacked malware as well, especially for various Telegram clippers. Other applications also appear, such as malicious repackaging of WhatsApp and Signal, as well as of crypto wallets and VPNs.

From June 2020 to January 2021, a malicious version of Telegram was available on Google Play under the name FlyGram, with over 5,000 installations. It was also available on the Samsung Galaxy Store and dedicated websites. Using the Android BadBazaar espionage

code, it can extract basic device information, contact lists, call logs and the list of Google Accounts. Surprisingly, it can only extract limited information and settings related to Telegram. Telegram's contact list and messages are not included unless users enable a specific FlyGram feature that allows them to back-up and restore Telegram data. This sends the back-up data to a remote server controlled by the attackers, giving them full access except for the collected metadata. [8]

From July 2022 to May 2023, there was a malicious version of Signal on Google Play called Signal Plus Messenger. It used the same Android BadBazaar espionage code and distribution channel as Flygram. Signal Plus Messenger seems to have been more focused on spying on communication than Flygram. It can extract the Signal PIN number that protects the Signal account, and it misuses the link device feature that allows users to link Signal Desktop and Signal iPad to their phones. [8]

Other Telegram-based malware on Google Play had descriptions in traditional Chinese, simplified Chinese and Uyghur, which told us a lot about the targeted region. It included full-fledged spyware capable of stealing the victim's entire correspondence, personal data, and contacts. The code of these malicious applications was only marginally different from the original Telegram code, which helped them pass the security checks to get onto Google Play. [9]

1.3 Summary

Repackaging attacks remain a significant and evolving threat in the mobile application ecosystem, particularly within the Android operating system. These attacks involve the modification of legitimate applications to include malicious payloads, often without the user's knowledge. As seen in the detailed cases presented, repackaging can lead to a variety of malicious outcomes, from stealing personal and financial information to distributing adware for revenue generation.

The prevalence of repacked malicious applications has been documented extensively, particularly during the early years of Android's rise. For instance, in 2012, the study "Dissecting android malware: Characterization and evolution" found that 86% of 1,260 malware sam-

ples were repacked versions of legitimate applications. [1] Similarly, in 2013, the study "AdRob: Examining the landscape and impact of android application plagiarism" identified over 44,000 cloned applications from a sample of 265,359, with significant concentrations in third-party markets. [3]

Recent trends indicate that despite enhanced security measures on official platforms like Google Play, attackers continue to find innovative ways to distribute repacked malware. This includes leveraging alternative distribution channels such as deceptive websites and Telegram channels. Noteworthy recent cases include the distribution of malicious versions of Telegram and WhatsApp aimed at stealing cryptocurrency, which employed advanced techniques such as clippers and optical character recognition (OCR).

The persistence of repackaging attacks highlights several critical points. Repackaging attacks have a broad scope, affecting a wide range of applications and posing severe risks to both personal and financial data. The impact on application developers is also significant, with substantial revenue losses due to application plagiarism.

Effective methods for detection and prevention of repacked applications are needed to at least keep them out of the reputable application stores. In order to maintain the security of platforms like Google Play, continuous collaboration between security researchers, application developers, and platform providers will be needed.

Social engineering is a highly efficient attack vector. Even if we manage to keep repacked applications out of reputable application stores, attackers can still successfully entice users to obtain these malicious applications from unofficial sources.

In summary, repackaging attacks remain a persistent threat, although significant progress has been made in identifying and mitigating them.

2 Observability and security tools

To detect repackaging attacks, it is not sufficient to know that the application was repacked. The application in suspicion often openly claims to be a modified version of an original, and sometimes such modifications are made with genuinely good intentions, non-malicious and even useful. The application must be caught red-handed to prove that it is indeed a case of repackaging attack.

There are two ways to investigate and detect malicious behaviour. Get the code for both the original and the modified application and compare them to see if there is any malware-like code in the modified part. Or observe the behaviour of the application and watch if it does something that the original application never does.

The first approach usually involves reverse engineering as the source code is often not available. Even if the source code for a modified application is available, it cannot be trusted as the attackers may include additional malicious code in a final build that they are not sharing. The biggest drawback is that the analysts have to be well-versed in the language in which the application is written, which might be a problem if they plan to investigate a wide range of different applications. During the lifecycle of an application, some parts of the code may even be rewritten in a completely different language. For example, Signal moved its protocol, cryptography, and the rest of its backend libraries from Java to Rust. [10]

The second approach attempts to identify anomalous or unexpected behaviour of the repacked applications. This includes monitoring system calls, network traffic, and file interactions that are critical to understanding the runtime activities of repacked applications. Unlike reverse engineering, which often involves static analysis of compiled code, observability tools allow us to dynamically monitor the execution of repacked applications. This allows us to observe the behaviour of the application in various contexts, including interactions with the operating system, network, and other applications.

However, it is most likely not feasible to observe and then analyse all aspects of the behaviour of the application. We need to select aspects of the behaviour that we want to

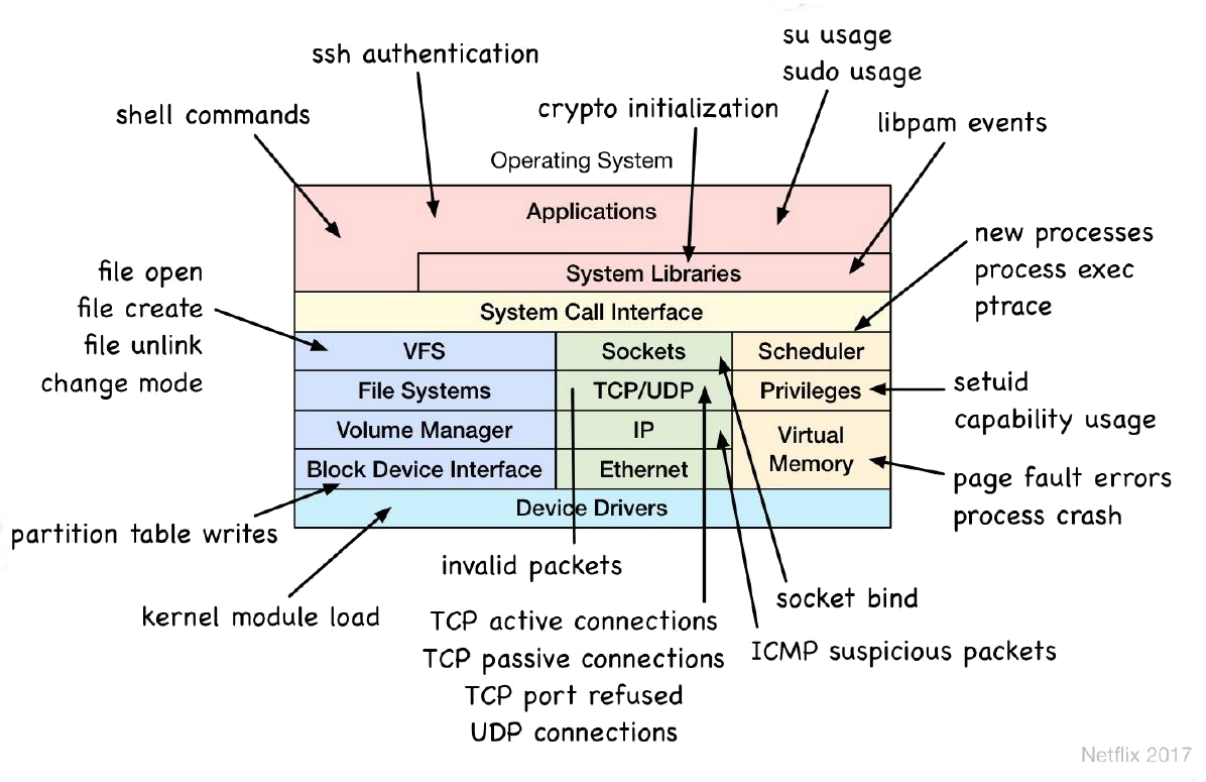

Figure 3: Security relevant traceable events [11], ©Brendan Gregg/Netflix, 2017

observe. Figure (3) is a diagram illustrating various system components that we might choose to monitor. For the selected components it also shows the types of activities or events that can be observed there. It depends on the application we are analysing which specific components and events we should focus on. However, checking the interaction with files, the internet connection and the processes used is a good starting point.

Traffic analysis is represented in the diagram by the green stack in the middle. It lets us identify communication patterns between repacked applications and external servers or command and control servers. Network traffic analyses can identify connections to malicious servers, data exfiltration attempts, or other network-based attack vectors. To limit overheads, it is important to track low-frequency events, especially in the context of security. Therefore, it is better to track TCP connection init rather than TCP send/receive.

VFS (Virtual Filesystem Switch) handles system calls related to file manipulations. By observing the VFS, we can see which files are being opened, copied or deleted. In this way,

we can find out whether the analysed application is reading files that it is not supposed to or is maliciously modifying configuration files.

The System Call Interface tells us when processes are launched. Here we can also observe when an application requests something from the kernel via a system call.

Watching the application layer itself, we can see shell commands, `ssh` authentication, crypto initialisation, `sudo` usage, `su` usage. The use of system libraries is easy to trace, as the `libpam` events have a good API for this. The use of cryptographic libraries might be interesting, as they may be used by ransomware or in decryption of obfuscated malicious code, but we need to keep in mind that most internet connected applications nowadays use cryptography for secure communication.

Below, we review some of the selected observability tools and explore some of their inner workings; then we look at the pitfalls of observability tools and the reasons why specialised security tools are needed.

2.1 Observability tools

In computer engineering, the word observability is used to describe the tools (for reading state), data sources (metrics and logs), and methods for understanding (observing) how a technology operates. Unlike benchmarks and other performance tools that change the state of the system to understand it, observability tools look at the system ideally without changing it. [12]

A perfect observability tool would just look at the system without touching it at all. However, real tools still have an impact on the system. Their execution consumes resources, usually negligible, but in some cases, it is enough to perturb the target of the study. [12]

We can split Linux observability tools into **tracing frontends** (the tool we actually interact with to collect/analyse data), **mechanisms for collecting data** for the frontends and **data sources** (where the tracing data comes from).

Tracing frontends are command-line or GUI applications in which the user types commands and receives output. A wide variety of them are listed in Figure (4) as the bold text next to

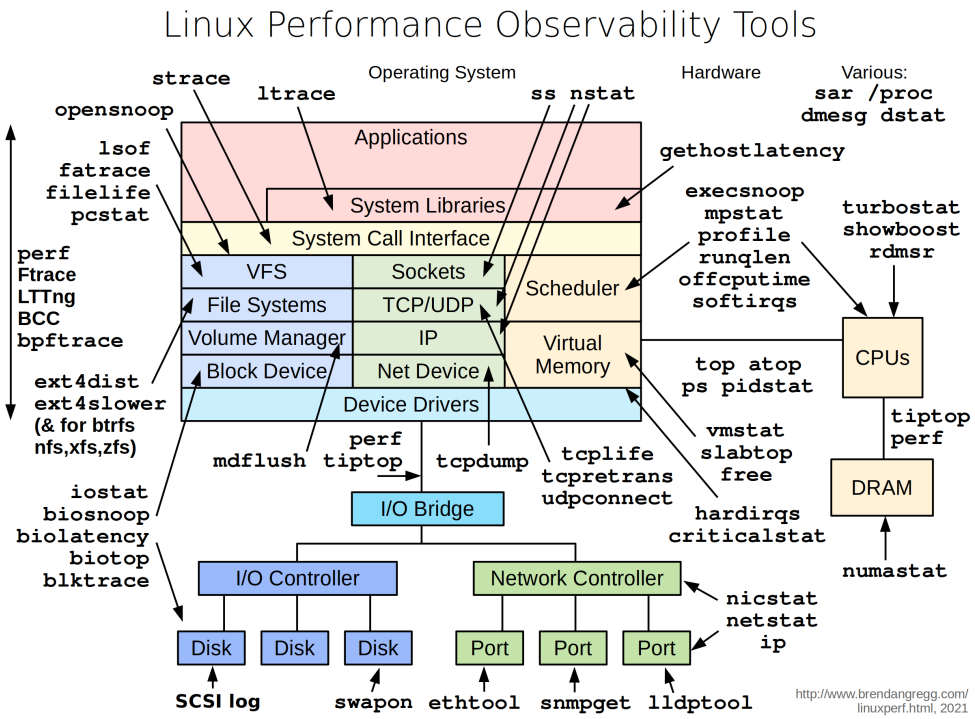

Figure 4: Linux observability tools, Brendan Gregg, 2021, CC BY-SA 4.0 International

the start of the arrows. As we can see from the diagram, each tool has its own specialised use and focuses on tracing a different part of the operating system. There are also some more general tools listed on the left of the diagram (perf, ftrace, BCC, bpftrac, LTTng) that can observe many parts of the operating system.

Mechanisms for collecting data get the data from the data source and serve them to the front end. Only `ftrace`, `perf` and `eBPF` are an official part of the Linux kernel, being directly integrated into the Linux kernel source code. While `LTTng`, `dtrace`, `SystemTap` or `sysdig` work as out of tree Loadable Kernel modules, which are developed and maintained separately from the Linux kernel and the kernel can load and link these Loadable Kernel modules at runtime. Beside `eBPF`, all the mechanisms for tracing data mentioned here share their names with their respective tracing frontend. eBPF frontends include BCC and bpftrace. Many of these mechanisms also use other mechanisms for collecting additional data, as you can see in Figure (5). For example, `perf` uses its own system call `perf_open()` as well as tracefs, a specialised file system used by `ftrace`. Many of these frontends are also currently working on implementing the use of eBPF.

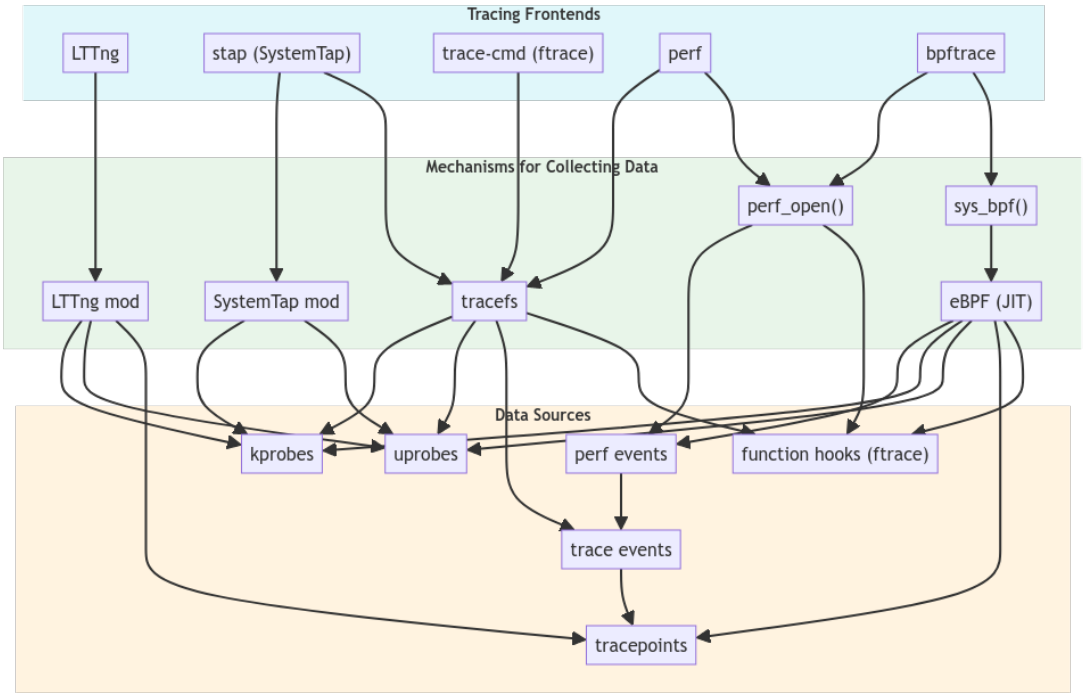

Figure 5: Tracing environment, figure by author, 2024

There are only four official data sources in Linux: kprobes, uprobes, tracepoints, perf_events. However, the way these mechanisms process the data sources can differ drastically, for example eBPF does all the data processing inside the kernel and only sends the final filtered result to the user-space, making it highly efficient.

First, we look at two commonly used tools. For observing the System Call Interface, we look at `strace`. For Net Device, we investigate `tcpdump` which is also used internally by Wireshark. After exploring some parts of their inner workings and related drawbacks, we look at some more efficient tools that use eBPF or perf.

For a more detailed overview of observability tools see [13] and [14] For a history overview see [15] and for minimal code examples see [16].

2.1.1 strace

`strace` is a system call tracer for Linux. It shows us most of the actions shown in Figure (4) under "System Call Interface". It was developed by Paul Kranenburg in 1991 for SunOS [17] and ported to Linux a year later.

A kernel feature known as `ptrace()` (process trace) is used by `strace` to pause the target process at the beginning and end of each system call, so that `strace` can read the state. The way in which `strace` works in the background, is also its biggest drawback. Pausing an application twice for each system call, and context-switching each time between the application and `strace` slows down the whole application considerably. [18] There is an infamous example of strace making the process 42 times slower. [19]

```
$ dd if=/dev/zero of=/dev/null bs=1 count=1M 2>&1 | grep -v records
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.668768 s, 1.6 MB/s


$ strace -f -qq -e signal=none -e trace=fchdir \
dd if=/dev/zero of=/dev/null bs=1 count=1M 2>&1 | grep -v records
1048576 bytes (1.0 MB, 1.0 MiB) copied, 28.0445 s, 37.4 kB/s
```

Thankfully, strace is still in active maintenance and evolving. Many improvements involve limiting `ptrace()` system calls or replacing them completely. An older but clear example of this can be seen in the year 2012, when strace started to use `process_vm_readv` instead of `PTRACE_PEEKDATA` to read data blocks such as filenames and data passed by I/O system calls. `PTRACE_PEEKDATA` gets one word per system call, which is very expensive. For example, in order to print the fstat system call, we need to perform more than twenty trips into the kernel to fetch one struct stat. The currently used `process_vm_readv()` can copy data blocks out of the process address space. For more examples of such small improvements, see [20].

A major speedup occurred in version v5.3 (September 2019) when Seccomp-assisted (Secure Computing Mode-assisted) system call filtering was added, which automatically generates and attaches a BPF program to filter system calls. This makes the execution of untraced system calls (those that we did not specify in the `strace` filter) two orders of magnitude faster. [19]

We get this speedup as system calls are allowed to run more or less normally with the use of `PTRACE_CONT` (ptrace continue). Only when the system call of interest appears, the traced process is stopped by the seccomp-stop. Then strace is called and the process is

restarted with PTRACE_SYSCALL. When the new `--seccomp-bpf` flag is used, the `dd` process is only 1.10 times slower than the same untraced process. Previously, without the `--seccomp-bpf` flag, tracing made the process 42 times slower than the untraced run. [19]

```
$ strace --seccomp-bpf -f -qq -e signal=none -e trace=fchdir \
dd if=/dev/zero of=/dev/null bs=1 count=1M 2>&1 | grep -v records
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.736511 s, 1.4 MB/s
```

However, in order to make further improvements to `strace`, `ptrace()` itself needs to be improved. This is problematic as already in 2010 the development of `ptrace()` was considered "frozen". No new features or improvements were added, only bugfixes. [21] Therefore, any improvement to `ptrace()` seems unlikely. There were also attempts to replace `ptrace()` completely with something new. This seems even more difficult as `ptrace()` is a standard function of the Linux kernel Applications Binary Interface. This cannot be removed from the kernel, unless "everybody" stops using it. This would mean that at first "everybody" would have to replace `ptrace()` with a working alternative – be it a standard tool such as `strace` or GDB, or as custom proprietary code used internally in some major companies, and only after that it could be removed from the kernel. This would require including an alternative to `ptrace()` in the kernel, while keeping `ptrace()` there as well. Doing so is possible, but it is a really unpopular option among Linux kernel developers. It would result in having to maintain two tracing interfaces instead of just one. [22]

This whole situation resulted in a deadlock in `ptrace()` development, which has come to an almost complete standstill. This would not be a problem if `ptrace()` was good enough, however, `ptrace()` is far from that. The `ptrace()` system call is considered to be one of the worse parts of the Unix interface due to its inelegant and complicated design. The design problems include strange semantics (it reparents the traced process to the tracer), so badly defined interactions with jctl (job control) that the current behaviours are broken to the point where achieving transparency with userland workarounds is impossible. For example, a task which is running under `strace` can be stopped with `^Z` as usual, but the shell will be unable to restart it as tracers currently have no way of knowing that the real

parent has tried to start a stopped process. [23], [24]

The general purpose of ptrace() is to allow one process to monitor and modify the state of another. It was originally intended for interactive debuggers such as the GDB debugger, but is now used in various sandboxing schemes, in `strace`, as well as for internal management of user-mode Linux. [22]

Whenever system calls have to work with the extended state within the kernel, the preferred mechanism for referring to this state in user-space is the file descriptor. With file descriptors there are well-defined mechanisms for event multiplexing, and thanks to them system calls behave "naturally". Unfortunately, `ptrace()` does **not** use file descriptors and instead depends on a rather arcane mechanism. A process to be traced is removed from its normal place in the process tree; the process doing the tracing becomes its new parent. In other words, `ptrace()` sets up a sort of temporary "foster home" for "children under scrutiny". The new parent can then learn about events in the child process through the `wait()` system call. [22]

In the Linux operating system, a process can only have one parent [25]. Because of this, any given process can only be traced by one other process at any given time. Usually this is not a problem as it is rare for someone to debug a process with two debuggers at the same time. However, this limitation often causes problems for developers of debugging tools. Because `strace()` is also used for sandboxing, when debugging a sandboxed process, it is not possible to use `strace` or a debugger. [22]

This limitation also allows processes to check if they are themselves being debugged or observed with `strace` or debugger. A malicious program can simply call `ptrace()` on its own malicious process. If `ptrace()` fails, the malicious program can simply assume that it is already being observed by `ptrace()` of someone else. Then, in order to avoid detection, the malicious program will refrain from any malicious actions during the time of observation. [26]

As `ptrace()` allows control over processes, it also has malicious uses on its own. It has already been used in several real-world attacks and exploitations. These include the

DirtyCow [27] bug exploitation, the remote access trojan called Puppy and CVE-2011-4327. In CVE-2011-4327, `ptrace()` allowed local users to obtain sensitive key information. [28]

Furthermore, the `ptrace()` system call is defined as a complex, multiplexer call (see the man page for details) that is hard to understand and difficult to use efficiently. User-space code which uses `ptrace()` tends to become encrusted with non-portable workarounds as `ptrace()` is hard to implement correctly and consistently. [22]

For more information about `strace`, we recommend the illustrated "strace zine" [29], and for a critical introduction with a list of many of the `strace` drawbacks, see [18], [30], [31].

`strace` is a great tool held back by the arcane `ptrace()`. For behavioural, analysis it might be problematic that `strace` is fairly easy to detect and cannot be used on many sandboxed applications. There are multiple alternatives to `strace` which use eBPF, such as `opensnoop`, or `perf` system calls and other tools, some of which we mention below. In the future, `strace` itself might start to use eBPF and solve most of its `ptrace` related problems with that.

2.1.2 tcpdump and Wireshark

tcpdump is a tool for capturing network traffic and performing packet analysis on live or captured traffic. It was developed in 1992 by Steven McCanne and Van Jacobson for SunOS as an alternative to the default `etherfind`, which was based on the Unix find command. Its user-space process was unable to process all incoming packets in real time, which would quickly overload the machine. `tcpdump` solved this problem by filtering packets with the in-kernel virtual machine BPF (see Section 2.1.3) developed alongside `tcpdump`. At the end of development, the compiler system and the filtering engine were put out of the `tcpdump`, and an API and reusable library called libpcap was created. A common file format called `pcap` was also created. It serves as an elaboration of the `tcpdump` flag `-w`, writes data into a file, such as `tcpdump -w http.pcap port 80`. [32]

Besides `tcpdump`, the most prominent project using libpcap is Wireshark. All Wireshark tools use libpcap for packet sniffing and can also use the `tcpdump` filter syntax as a capture filter. [33] In addition, Wireshark offers a GUI, more advanced filtering,

and other features such as connecting TCP packets from the same connection. As Wireshark uses the same library as `tcpdump`, it is possible to capture packets with `tcpdump -c 100 -w my_packets.pcap dest port 8080` and then later open them with Wireshark using `wireshark -k -i -`. This might be especially useful in case we are capturing packets on a remote host where Wireshark is not installed. Here a command such as `ssh some.remote.host tcpdump -pni any -w - -s0 -U port 8888 | wireshark -k -i -` can be used.

If for some reason you do not want to or cannot use `tcpdump`, but still want just a command-line application with no GUI, Wireshark also comes with a collection of terminal tools. There is a TUI version of Wireshark called `tshark`. Both `tshark` and Wireshark have mostly equivalent functionality. However, only in Wireshark's GUI it is possible to select which packets to save, `tshark` will record everything. Then `tshark` itself runs another command-line tool from the Wireshark developers called `dumpcap`, which is a small program whose sole purpose is to capture network traffic, while retaining advanced features like capturing to multiple files. In most cases, you will just use a `tshark` command instead of a `dumpcap` command, [34] but as `tshark` only runs `dumpcap`, their performance should be the same. [35]

For more information about Wireshark and tcpdump, see [36].

2.1.3 eBPF

BPF was developed by Steven McCanne and Van Jacobson alongside `tcpdump` as its Kernel module, a virtual machine model that would run in the kernel. Now, eBPF makes the Linux kernel programmable without the need to modify the kernel code or create Kernel modules. Thanks to this, eBPF tools form the basis of many versatile and efficient observability tools such as BCC, bpftrace or Cilium. [37]

In 1992 BPF was introduced as BSD Packet Filter, a pseudomachine that can run filters to determine whether to accept or reject a network packet. Filters for BPF are written as programs using the BPF instruction set, a general-purpose set of 32-bit instructions that closely resembles an assembly language. [38], [39]

Bytecode as written in the 1992 article The BSD Packet Filter: A New Architecture for User-level Packet Capture

```
ldh     [12]
jeq     #ETHERTYPE IP, L1, L2
L1:     ret     #TRUE
L2:     ret     #0
```

code output of `sudo tcpdump -d -i wlan0 -n ip`, `-d` flag outputs the bytecode.

```
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 3
(002) ret      #262144
(003) ret      #0
```

Both of these examples of instruction filter out packets that are not Internet Protocol (IP) packets, leaving only IP traffic in the output. The first instruction `ldh` loads a 2-byte value starting at byte 12 in an Ethernet packet. The instruction `jeq` compares this 2-byte value with the value that represents an IP packet. If it matches, the execution jumps to the instruction labelled `L1`, and the packet is accepted by returning a nonzero value `#TRUE`. If it does not match, the packet is not an IP packet and is rejected by returning `0`. [39], [40]

As BPF's name implies, the BPF implementation came from BSD under a BSD license and was used as a packet filter. Packet filters make network monitoring more efficient by discarding unwanted packets in the kernel and copying to the user-space only the data that the user is actually interested in. BPF has a register-based filter evaluator with a non-shared buffer model. It was intended as a replacement for Unix's default stack-based filter evaluator.

Later BPF started to be called Berkeley Packet Filter (BPF). Nowadays, in order to avoid confusion with the newer version of BPF, it is also called classic BPF (cBPF). [39], [41]

cBPF was included in Linux in 1997 in version 2.1.75. For a long time it was only used as a socket filter by the packet capture tool tcpdump (via libpcap). [42]

Using the same principle of doing most operations in the kernel and only sending the final result to user-space could speed up many other processes besides packet filtering. However, for almost 15 years, the in-kernel virtual machine went unnoticed. cBPF capability to run untrusted programs in a privileged context was underutilised, even though it provided a secure alternative to the Kernel modules.

Things started to change in 2011 when a just-in-time (JIT) compiler was added to cBPF on x86. This made customised network-packet filtering extremely fast. [15], [43]

The first non-networking use case (probably even the first outside of libpcap) appeared in 2012, in Linux kernel version 3.5, when `seccomp-bpf` was introduced. It is a system call filtering program that uses a configurable policy implemented through BPF instructions. For example, it is used in the Linux version of Chrome as the main layer-2 sandbox, designed to shelter the kernel from malicious code executing in the userland. [42], [44] `seccomp-bpf` is an improvement upon `seccomp` (SECure COMPuting Mode), which was introduced in the Linux kernel in version 2.6.12 (on 8 March 2005). While `seccomp` restricts the system calls available to a process to only `read`, `write`, `_exit` and `sigreturn`, `seccomp-bpf` uses BPF programs to filter arbitrary system calls and their arguments (constants only, no pointer dereference). [45] Additionally, according to the Linux Kernel documentation "BPF makes it impossible for users of seccomp to fall prey to time-of-check-time-of-use (TOCTOU) attacks that are common in system call interposition frameworks". [46] After that, BPF was also used in the iptables module xt_bpf, available since Kernel 3.9 [47], and in the "cls_bpf" classifier for traffic shaping (QoS), available since Kernel 3.13 [48], [49]. The use of BPF was slowly gaining traction.

Developer Alexei Starovoitov created extended (eBPF) in 2014 to make BPF programs applicable to other parts of the kernel, including tracing. [15]

In March 2014, eBPF replaced classic BPF (cBPF) in-kernel when it was accepted by David S. Miller, the primary maintainer of the Linux networking stack. Since then, the Linux kernel internally translates classic BPF (cBPF) calls into the eBPF instructions in order to provide backward compatibility. [50], [51]

The eBPF brought major changes compared to the classic BPF. The BPF instruction set was completely overhauled to be more efficient on 64-bit machines; the interpreter was entirely rewritten; the eBPF verifier was added to ensure that eBPF programs are safe to run; a new data structure for sharing information between BPF programs and user-space called 'maps' was introduced; the `bpf()` system call was added to the user-space; and many other changes were applied. In 2015, the ability to attach eBPF programs to kprobes was also added. [40]

This extended use of eBPF beyond packet filtering also led to a formal name change, where eBPF is no longer considered to be an acronym, but it is used as a proper name, sometimes even just referred to simply as BPF.

An interesting topic, beyond the scope of this thesis, is why Linux kernel developers decided to reinvent the whole infrastructure of virtual machines, JITs and scripting languages when well-established solutions already existed outside the kernel world. For example, there were active initiatives in the past to get Lua into the kernel. [52] The answer to why eBPF has succeeded where these other initiatives have not may lie in the fact that eBPF is based on a component that has been in the Linux kernel since 1997. And it is much easier to persuade kernel developers to allow the extension of this already well-tested and trusted tool than to introduce a completely new thing into the kernel.

It is also important to remember that eBPF, like any other observability tool, is not perfect and has many flaws, such as being susceptible to the time-of-check to time-of-use issue and other problems which we further describe in Section 2.2. In addition to these problems, there are eBPF specific problems. For example, the kernel sometimes fails to fire eBPF probes, which completely prevents eBPF from observing an event it is supposed to log. You can read more about these eBPF problems in [53].

To make use of eBPF, you need BPF bytecode that tells the virtual machine what to do. Thankfully, there is usually no need to write it directly, but there are multiple front-ends with their own scripting languages.

The best start for beginners is `bpftrace`, which is suitable for one-liners and short scripts.

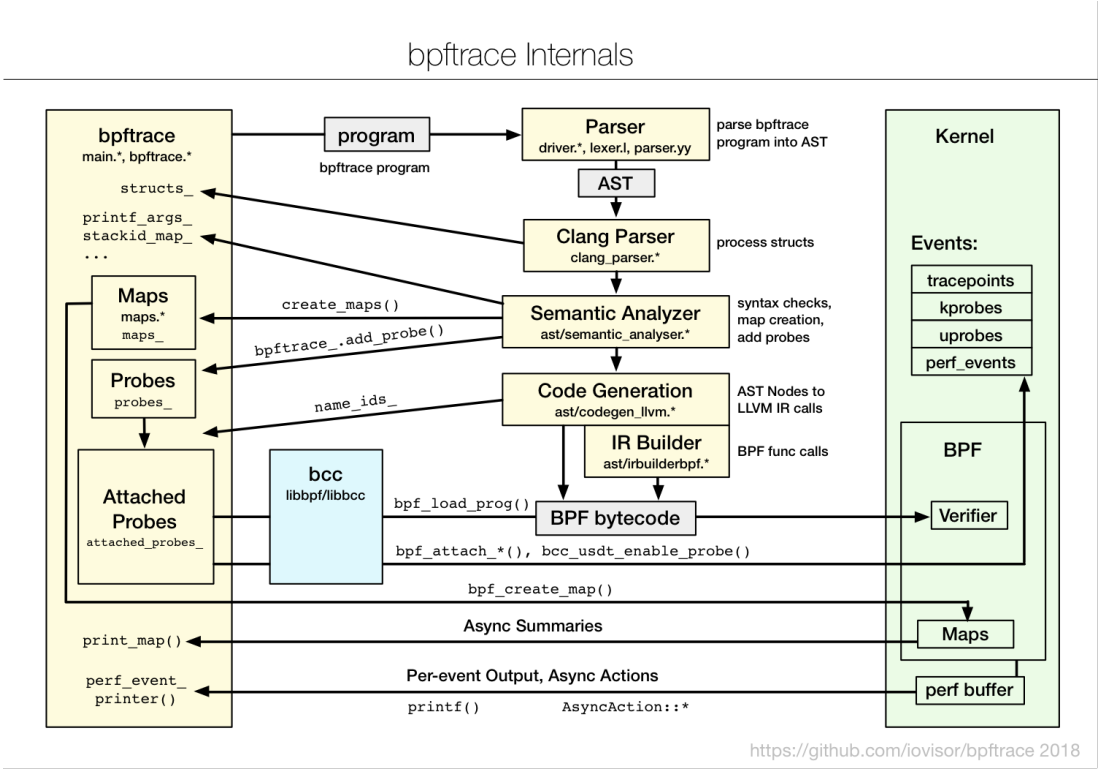

Figure 6: Internal structure of bpftrace, Brendan Gregg, 2018 [54]

For more complex tools and daemons, BCC is recommended. Libraries from BCC are also used internally by `bpftrace` as we can see in Figure 6.

For many generic tasks, there is no need to write your own script as many are already readily available, especially for BCC as shown in Figure 7.

To learn more, see https://brendangregg.com/ebpf.html and https://andreaskaris.github.io/blog/networking/bpf-and-tcpdump/. For a full timeline, see https://en.wikipedia.org/wiki/EBPF#History. These two videos may also be helpful to learn more about the history of BPF. Videos https://www.youtube.com/watch?v=DAvZH13725I Kernel Recipes 2022 – The untold story of BPF eBPF and Kubernetes: Little Helper Minions for Scaling Microservices – Daniel Borkmann, Cilium https://www.youtube.com/watch?v=99jUcLt3rSk

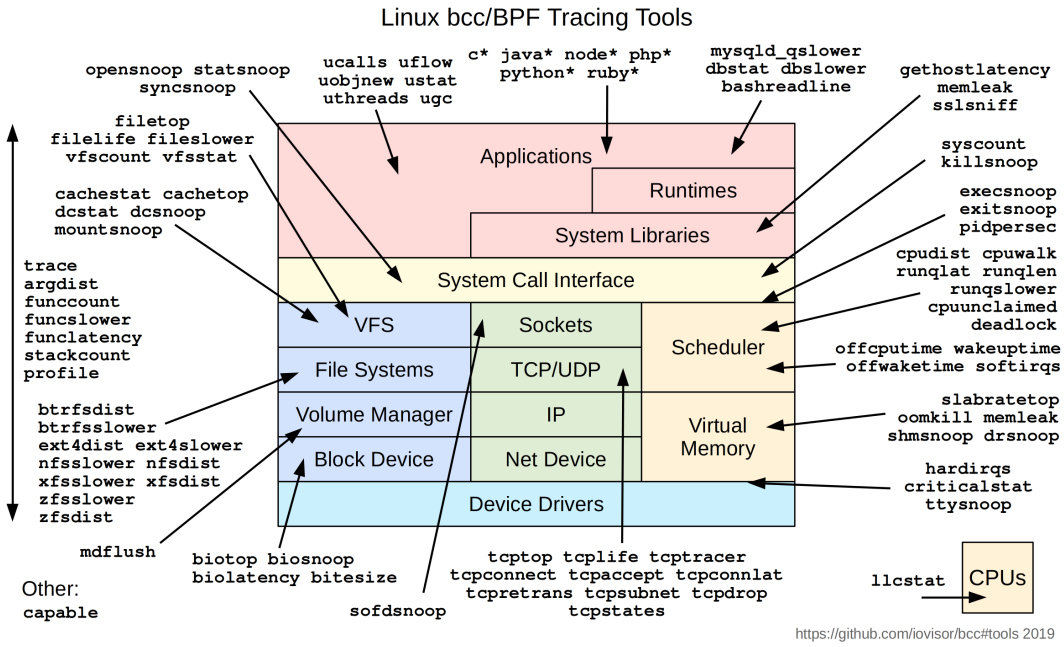

Figure 7: BCC tools, Brendan Gregg, 2019 [55]

2.2 Pitfalls of observability tools

The problem with the tools mentioned above is that they are designed to be observability tools, not security tools. While they are useful for a quick initial analysis, they are not perfect for security monitoring and security analysis. They are designed to have the lowest possible overhead because they are used for analysing live production systems, which introduces many trade-offs. Detection by observability tools can be avoided by overwhelming the system, time-of-check-time-of-use (TOCTOU) attacks and other techniques. [56]

Even simple `ls` can be avoided by including escape characters such as `\n` newline character, carriage return `\r` or backspace `\b` into a filename. `top` relies on the "comm" field (the command name) to display process information. An attacker might change the names of malicious processes to resemble innocent or common process names to avoid detection. `tcpdump` will drop packets when the system is overloaded, resulting in incomplete visibility. An advanced attacker could overwhelm the system with mostly innocent packets, or wait for the moment when the system is overwhelmed, and only send malicious packets at that time. In a similar way, `strace` could be overwhelmed by a simple fork bomb `:(){ :|:& };:` or by injecting code into legitimate processes or libraries.

As we are comparing a suspicious application with its legitimate version, most basic techniques of simply overwhelming the system would automatically raise a flag, but it is good to keep this in mind. In order to secure strict non-repudiation, it is possible to set the kernel to halt immediately if it fails to log an event. There are also less drastic ways, such as at least logging the failure to log as well as specialised security tools better suited for such analysis. Adapting observability tools into security tools generally increases overhead, for example by adding extra probes. Good security tools need to be built differently. Ideally, a good security tool should be integrated directly into the Linux Security Modules framework, use a plugin model instead of standalone CLI tools, support configurable policies for event drop behaviour, have optimised event logging, and more. [56]

Brendan Gregg writes in his blog "eBPF Observability Tools Are Not Security Tools" [56]:

> "Had I written these as security tools to start with, I would have done them differently: I'd start with LSM hooks, use a plugin model instead of standalone CLI tools, support configurable policies for event drop behavior, optimize event logging (which we still haven't done), and lots more."

Afterwards, he mentions that none of this should be new to experienced security engineers, as it should really be the basis of what we should expect from a security tool. Below we try to expand on these brief notes and provide some more information on each of the points mentioned.

Linux Security Modules hooks are used to add extra security checks and controls to the Linux kernel beyond what is provided by the base kernel. These security hooks are integrated directly into the Linux Security Modules framework, they leverage the existing security infrastructure in the Linux kernel and integrate the security tool more tightly with the system's security mechanisms.

The plugin model involves designing the security tool to be extensible through plugins or modules. Instead of a monolithic standalone tool, a plugin model allows users to add or customise functionality through separate modules. This approach provides flexibility by allowing users to choose the specific features or modules they need, and it allows for easier

maintenance and updates.

Configurable policies for Event Drop Behaviour allow users to define rules for the case when certain events should be dropped or allowed. For example, users might want to configure the tool to drop specific types of events under certain conditions and allow them under different circumstances. This can help mitigate attacks aimed at overwhelming system resources, obfuscating logs, or masking malicious activities. To defend against overwhelming attacks administrators can set thresholds for the number of events per second or per minute. If incoming events exceed these thresholds, the system can be configured to drop or ignore excessive events, preventing the attacker from overloading the logging system.

Optimising event logging involves minimising the impact on system performance, implementing mechanisms to handle high volumes of events without losing critical information, and ensuring that the logs are appropriately detailed.

At first glance, observability tools could be immediately reused as security tools. While observability tools can provide a great deal of useful information and sometimes even detect malware, these tools are easily subject to targeted evasion techniques.

2.3 Summary

Observability tools are invaluable for detecting repackaging attacks, because they provide dynamic insights into the behaviour of applications in real-time. Unlike static analysis, which examines the code of applications, observability tools focus on monitoring the runtime behaviour, interactions, and system calls of applications to identify any anomalies or unexpected activities. This chapter has highlighted the significance of observability tools in security and provided an overview of the methodologies and specific tools used for monitoring and analysing application behaviour.

We explored system call monitoring with strace, network traffic analysis with tcpdump and Wireshark, and the up-and-coming eBPF. `strace` is a widely used tool for tracing system calls, providing detailed insights into the interactions between applications and

the operating system. However, it can be slow due to its reliance on `ptrace()`. tcpdump and Wireshark capture and analyse network packets, which can help identify malicious connections and data exfiltration attempts. Wireshark, with its GUI, offers more advanced filtering and analysis capabilities, which are to some extent also available in its TUI version, `tshark`.

The evolution of eBPF has significantly enhanced the capabilities of observability tools. eBPF allows for efficient in-kernel processing of monitoring tasks, reducing the performance impact, and providing more granular insights into the behaviour of the system. Tools like `bpftrace` and BCC leverage eBPF to offer advanced monitoring capabilities, making them valuable assets in detecting and analysing repacked applications.

However, it is important to recognise the limitations of using observability tools for security purposes. Observability tools are primarily designed for performance monitoring and debugging, not security. Their focus on low overhead can make them susceptible to attackers' evasion techniques. Techniques like overwhelming the system or using sophisticated evasion strategies can bypass detection by these tools. This highlights the need for specialised security-focused tools. In order to assure robust protection against repackaging attacks, our tools should ideally be integrated directly into the Linux Security Modules framework and equipped with configurable policies for event handling.

While observability tools provide essential insights into application behaviour and are crucial for detecting repackaging attacks, they must be complemented with specialised security tools to address their limitations and enhance overall security effectiveness.

3 Previous analyses of Signal and Telegram

Most of the findings about what is the legitimate behaviour of the instant messaging application comes from studies that with use of device and network forensic methods tried to learn something about the legitimate applications. They tried to access data and metadata from often encrypted databases and determine the type of activity based on visible network traffic. Their aim was not to directly assess the legitimate behaviour of the application, they assumed that they work with the legitimate version. They were undertaking a planned forensic analysis in which they tried to identify the location in the file system used by the application, the servers that the application connects to and other data useful for us.

Most of the available studies focus on device forensics [57], with the main focus on the internal structure of databases stored on devices, but also include analysis of the file structure and extraction of useful information from artifacts. The analysis of encrypted file structures and databases is less studied. [58]

Network forensics traces communications through the network packets to identify the traces of the application's activity. For instant messaging there are already studies that can identify connection establishment, an encryption protocol, and payload sizes for analysis, and some can also distinguish behaviour such as calling, texting and similar.

For a more complex literature overview, we recommend [59], [60].

Unsurprisingly, most of the available studies focus on WhatsApp, as it is the most widely used instant messaging application. There are also numerous Telegram studies available [61]. The number of forensic Signal studies is surprisingly low.

Research on Telegram has covered various aspects, including device forensics across different platforms.

The 2017 study on Telegram for Windows Phone provided a comprehensive overview of the application's architecture and user data file management, which remains relevant beyond the discontinued Windows 10 Mobile platform. It included a useful description of general

concepts related to Telegram, such as the TL (Type Language), names of user data files, as well as a detailed guide on how to get information from these data files. [57] Most of the information in the paper seems to be applicable for other platforms and is therefore useful, even though Windows 10 Mobile is no longer supported since 10 December 2019 [62] and Microsoft is not developing another mobile operating system. [63]

Another study in 2017 analysed network traffic from Telegram, as well as Facebook and Twitter, on the now-obsolete Firefox Mobile OS, revealing insights into the application's server connections and data transmission patterns. Firefox Mobile OS simulator was used. The generated traffic was sniffed by Wireshark, resulting in a list of IP addresses, ports, domains and subdomains used by these applications. Key takeaways for Telegram are that when generating the registration key, Telegram connects to `us-west-2.compute.amazonaws.com` (the analysis was done in the USA, so the domain name ends in `us-west-2`, this would change depending on the location), which was used to push and generate the registration key for Telegram, and, unsurprisingly, to `telegram.org`. After registration, Telegram mostly connects just to `telegram.org`, but also receives traffic from `github.map.fastly.net` when it receives a message from another contact. When sharing location with contacts, traffic comes from IPs associated with Google Internet Authority, and when playing a song received from another contact, Telegram connects to `.us-west-2.compute.amazonaws.com`. [61]

A 2017 investigation into Telegram on a virtualised Android device unearthed significant forensic data, determined the structure and format of these artifacts, implemented the corresponding decoding procedures in a Java program, and mapped the data stored by Telegram Messenger to the user actions that generated it. Using the above mapping, they also show how to recover the account used with Telegram Messenger and how to reconstruct the contact list of the user, the chronology and contents of both textual and non-textual messages, and the log of voice calls made or received by the user. It also offers to serve as a detailed template for similar forensic analyses thanks to its thorough and well readable layout. [64]

The 2018 research compared data extraction methods on Telegram, WhatsApp,

Viber and WeChat across both rooted and un-rooted Android devices, uncovering the specific file storage practices of Telegram and other messaging applications. The Android Debug Bridge (adb) commands `pull` for rooted phones or `backup` for un-rooted phones were used to extract the data. Telegram files were only accessible on rooted devices with the use of `pull` command. It was found that Telegram in version 4.5.1 stored an encrypted SQLite database of chat messages at `/data/data/org.telegram.messenger/files/Cache4.db`, details about the account used at `/data/data/org.telegram.messenger/shared prefs/userconfing.xml`, profile photo `/data/media/0/Android/data/org.telegram.messenger/cache` and copies of files sent and received at `/data/media/telegram`. The paper also offers a brief description of different kinds of chats available in Telegram and the encryption used. [58]

The forensic analysis of Signal is an understudied field. The 2021 study offers an extensive traffic analysis of Signal on an Android phone, with traffic capture and filtering performed by a dedicated firewall. The study revealed the obscured design of the Signal applications and was able to identify different activities of the Signal application such as calls, text or typing indicator. [59]

Since the publication of [59] many domain names and other information mentioned in it have already changed. This highlights the fact that forensic analysis of instant messaging is a dynamic field. The results of the analysis for any application will not stay up-to-date for long, and therefore an efficient way of creating a new analysis is needed.

4 Analysis of Signal and Telegram

We used a wide variety of tools described in Chapter 2 in order to analyse the applications, detect key patterns of both Signal and Telegram, and uncover suspicious activities.

As the first step, we wanted to identify which patterns are specific for Signal and Telegram, and which just occur because of the third-party software that they use. To do this, for both Signal and Telegram, we compared the official version of each application with an example demo of their GUI framework, as well as with the TUI version of the instant messaging application which strips it down to its core.

For Signal, we compared the official `signal-desktop`, the unofficial `signal-cli` which uses the Java library `libsignal-service-java` from the official `Signal-Android` repository [65] and `electron-quick-start-typescript` [66], because `signal-desktop` uses the TypeScript version of Electron for its GUI.

In the case of Telegram, we compared the official `telegram-desktop`, and the example notepad project from the Qt documentation as `telegram-desktop` uses Qt as its GUI framework. [67] This helps us to differentiate between the core footprint of Telegram itself and the more general footprint of Qt based applications.

4.1 Details of testing

The official Signal for Linux, called `signal-desktop`, uses the Electron framework as the GUI, which uses the Chromium browser engine in the background. The `telegram-desktop` uses the Qt framework for its GUI.

In the case of `signal-desktop`, we tested version 7.0.0-1, which we got from the stable branch of the Manjaro repository [68], [69].

signal-cli is a command-line interface for the Signal messenger. The intended use of signal-cli, as claimed by the developer, is to be used on servers to notify system administrators of important events, but there are also multiple TUI applications available. It occupies an important niche, as it is the only unofficial command-line interface for the Signal messenger,

and the official Signal developers do not provide one.

signal-cli is based on custom patched libsignal-service, the official Signal-Android source code, which also includes Signal's official platform agnostic libsignal, which handles all cryptography and protocols. [70]

We tested v0.13.3 of signal-cli, specifically `signal-cli-0.13.3-Linux-native.tar.gz`, which we got from the official developer's repository. [71]

In both `signal-desktop` and `signal-cli` we tested receiving and sending messages. For `signal-desktop`, we also tested audio calling.

For `signal-desktop`, we started the tracing tools, then opened signal-desktop with the command `signal-desktop`, and after that we received and sent messages inside the GUI.

With `signal-cli` we traced and performed all these operations separately, because the command-line interface allows this easily. In the logs, we replaced the real numbers with randomly generated fake ones, where `+421 909 104 930` is the phone number of the user of signal-cli, and `+421 909 543 173` is the phone number of another user with whom we are interacting. We first linked our device with the main Signal application on the Android phone with `./signal-cli link -n +421909104930` (this also needs to be done on signal-desktop, but we did not capture this process there), `./signal-cli -u +421909104930 receive` to get the list of contacts and groups from the main device, `./signal-cli -u +421909104930 send -m "Hello, how are you?" +421 909 543 173` to send the message to `+421 909 543 173` and `./signal-cli -u +421909543173 receive` to check if there are any new messages from `+421 948 413 685` and receive them if there are some.

When testing `telegram-desktop`, we used version 5.1.5-1 which we got from the stable branch of the Manjaro repository. We traced receiving and sending messages, and we also traced opening of the video files inside Telegram.

Having this wide range of software, we tried various tracing tools on it in order to monitor different aspects of its behaviour.

Below we present the output from `strace` and `Wireshark`, which was used for the pre-

liminary analysis of the behaviour. We also experimented with `gdb`, `ltrace` and `perf`, mainly in order to get more information about network and DNS related calls such as `getaddrinfo` and `getnameinfo`. However, this did not bring us any new data. Finally, we tried to redo the very same observations that we did with `strace` and `Wireshark` just with BCC.

We also compared the results of our observations with AppArmor profiles available from the AppArmor.d project. [72]

4.2 strace

When using `strace`, we tested both Signal and Telegram.

For Signal, we tested the Electron GUI with `strace -f npm start`, the official Signal with `strace -f signal-desktop` and the bare bones TUI version of Signal with `strace -f -o signal-cli_command_name ./signal_cli command`.

For Telegram, we tested the Qt GUI with `strace -f -o qt_strace ./build-application-Desktop-Debug/application`, and the official Telegram application with `strace -f -o telegra_strac telegram-desktop`.

To make sense of the raw strace output, we filtered it out with Python scripts. First, the `path_exec_summary.py` script filters out the list of all file paths mentioned in the output and also filters out the system calls. In both cases, the script outputs a json file, where the string key is the file path or system call name, and the integer value is the number of occurrences of the given key in the file.

Then we used the `compare_json.py` script, which takes two json files as input and outputs `common_{file1_name}-AND-{file2_name}.json`, where string key is again the file path or system call name which occurs in both files and value is a list with two integer values, where the first value is how many times the key occurs in `file1` and, the second value is how many times the key occurs in `file2`. The other two outputs `IN-{file1_name}-NOT-IN-{file2_name}.json` and `IN-{file2_name}-NOT-IN-{file1_name}.json` have the same string key and integer value as those from

path_exec_summary.py.

4.2.1 signal-desktop and signal-cli

signal-desktop and signal-cli limit their access pattern within the /home/ directory, predominantly engaging with their respective configuration files and directories. This makes it easier to detect any anomalous or suspicious activities within more personal directories, such as /home/jackie/Documents/. Both signal-desktop and signal-cli specifically access the directory /home/jackie/.local/bin/uname.

Both applications have their own directories inside the /home/ directory, where they store their data. For signal-cli, this is /home/jackie/.local/share/signal-cli for storing avatars and account data, showcasing its detailed management of user-specific data. In the case of signal-desktop, it is /home/jackie/.config/Signal directory, which handles SQLite databases and various configuration files related to its operation.

Here we can also see how signal-cli and signal-desktop handle database operations. Signal-cli accesses various files in /home/jackie/.local/share/signal-cli/data, including account.db, account.db-journal, and account.db-wal, among others. In contrast, signal-desktop handles its database operations in /home/jackie/.config/Signal/sql, interacting with db.sqlite, db.sqlite-shm, and db.sqlite-wal files. This difference is because signal-desktop uses the Electron package better-sqlite3 in order to handle sql, so signal-cli has to use different methods as it is not Electron based. signal-desktop also accesses more system-oriented directories, including the font configuration cache in /home/jackie/.cache/fontconfig/, indicating its interaction with system resources for rendering and display purposes.

Both signal-desktop and signal-cli access common folders and files in /home/. They both attempted to open files using the openat(AT_FDCWD, {path}, O_RDONLY|O_CLOEXEC) system call, where the paths included "/run/systemd/machines/cdn.signal.org" and "/run/systemd/machines/chat.signal.org", both resulting in -1 ENOENT (No such file or directory). Additionally, they both made the connect(21, {sa_family=AF_UNIX, sun_path="/run/systemd/resolve/io.systemd.Resolve"},

42) call, which similarly returned -1 ENOENT (No such file or directory). These operations indicate attempts to access specific system resources that were not available.

Beside /home/ both applications were accessing paths involving /dev/, /etc/, /proc/, /run/systemd/, /usr/lib, /usr/bin/, /usr/local/, and /tmp/ which are common system paths that are not unique to Signal's functionality.

Additionally, both applications utilise network-related system calls such as connect, recvmsg, and sendto, which are essential for transmission and receiving of messages. They both also use getrandom for generating secure random numbers, which is not surprising as they both use end-to-end encryption.

In summary, the shared footprint of signal-desktop and signal-cli is evident in their access to specific directories, handling of user-specific database files, and utilisation of network and security-related system calls. This combination of operations and file accesses underscores their core functionalities centred around secure messaging and data management within the Signal ecosystem.

4.2.2 signal-desktop and electron-quick-start-typescript

Both signal-desktop and the minimal example Electron application, electron-quick-start-typescript, create configuration directories within /home/user_name/ .config/program_name (/home/jackie/.config/Electron for the Electron Quick-start and /home/jackie/ .config/Signal for signal-desktop). However, signal-desktop populates this directory with several unique files and subdirectories that are not typically found in other Electron applications by default. Specifically, signal-desktop includes the following files and directories:

"SingletonCookie"

"SingletonLock"

"SingletonSocket"

"sql/db.sqlite"

"sql/db.sqlite-journal"

"sql/db.sqlite-shm"

"sql/db.sqlite-wal"

"attachments.noindex"

A special case are directories with a variable path structure where only a small part is different. These file path patterns are shared by both applications as they are core part of Electron's behaviour, for example `/dev/shm/.org.chromium.Chromium.xxxxxx`, where `xxxxxx` represents a sequence of alphanumeric characters. Beside `.org.chromium.Chromium.xxxxxx`, these shared paths also include `Local Storage/leveldb/xxxxxx.log`, and `Session Storage/xxxxxx.log`, where `xxxxxx` is a string of numbers starting with zeros. Additionally, both have `blob_storage/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`, where `x` is either a letter or a number, as well as several `/proc` paths such as `/proc/xxxxx`, `/proc/xxxxx/exe`, `/proc/xxxxx/stat`, `/proc/xxxxx/status`, `/proc/xxxxx/statm`, `/proc/xxxxx/oom_score_adj`, and `/proc/xxxxx/task/xxxxx/status`, where `x` represents numeric values.

Furthermore, both applications include the path `/usr/lib/.../node_modules/@module_name/...`, reflecting their shared use of Node.js modules.

As expected, there were system calls used exclusively by `signal-desktop`. In total 16 additional system calls occurred. Out of these calls, the most interesting were `symlink` and `chmod` used for file operation. Interestingly, there were also 6 system calls used only by the `electron-quick-start-typescript`. This was not expected as the Electron Quickstart application has no package dependencies and has the same development package dependencies as signal-desktop. These system calls might be utilised during the `npm run build` process, which is invoked at the start of the `electron-quick-start-typescript`, but not at the start of `signal-desktop`.

4.2.3 `telegram-desktop` and qt example application

Telegram exhibits distinct file paths and behaviours that remain identifiable even when files and directories are systematically renamed to obfuscate their origins. One such definitive path is the `tdata` directory located within `/home/jackie/.local/share/TelegramDesktop/tdata`, which is a characteristic directory used by Telegram for data

storage.

Telegram tries to access specific files in the `/usr/bin/` directory, including:

```
/usr/bin/TelegramAlpha_data
/usr/bin/TelegramBeta_data
/usr/bin/TelegramForcePortable
```

Interestingly, these files are not available on the system.

Moreover, Telegram's system interactions reveal distinct patterns, such as the frequent access to systemd-related paths and libraries, as demonstrated by the following system calls:

```
openat(AT_FDCWD, "/usr/lib/libsystemd.so.0", O_RDONLY|O_CLOEXEC) = 3
access("/run/systemd/journal/socket", F_OK) <unfinished ...>
openat(AT_FDCWD, "/run/systemd/machines/mozilla.cloudflare-dns.com",
O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
connect(49, {sa_family=AF_UNIX,
sun_path="/run/systemd/resolve/io.systemd.Resolve"},
42) = -1 ENOENT (No such file or directory)
```

These calls highlight Telegram's interactions with systemd components, such as accessing libraries and sockets related to systemd services. Additionally, Telegram makes network requests predominantly visible through attempts to connect to specific systemd-resolved paths, though it primarily connects to predefined IP addresses without making extensive additional requests.

Further analysis of the file path data reveals more Telegram-specific entries, such as:

```
"/home/jackie/.local/share/TelegramDesktop/tdata/settingss": 92,
"/home/jackie/.local/share/TelegramDesktop/tdata/D877F783D5D3EF8Cs": 60,
"/home/jackie/.local/share/TelegramDesktop/tdata/D877F783D5D3EF8C/maps": 55,
"/home/jackie/.local/share/TelegramDesktop/tdata/26B970315066C192s": 42,
```

Here even if "TelegramDesktop" is replaced by another name, it can stay clearly identifiable

as Telegram thanks to the distinct naming of files and directories, such as “settingss”, which is a less standard name than just “settings”. Similarly, 16 hexadecimal numbers and after them ‘s’ or “/maps” is a specific naming convention.

These entries reflect additional directories and files used by Telegram for configuration and data storage, further distinguishing its footprint from other applications.

From the comparison data, another potential path defining Telegram’s unique behaviour is the frequent use of certain system calls, such as `epoll_wait`, `epoll_ctl`, and `timerfd_settime`. These system calls are more prevalent in Telegram compared to a simple Qt example program. The notable system calls include:

```
"epoll_wait": 458,
"epoll_ctl": 242,
"timerfd_settime": 225,
"setsockopt": 203,
"getsockopt": 127,
"memfd_create": 20,
"ftruncate": 20
```

These system calls indicate extensive use of event handling, timer management, and socket options configuration, which are critical for Telegram’s real-time messaging functionality.

None of these alone can be used as definitive identifiers, however, when they all appear together, they make it probable that the investigated application is using Telegram in the background.

4.2.4 Comparison of strace analysis with Signal and Telegram AppArmor profiles

AppArmor is a Linux Security Module implementation that restricts the capabilities and permissions of an application with profiles that are set on a per-program basis. It provides a Mandatory Access Control (MAC) like security system for Linux. AppArmor complements the more traditional UNIX model of Discretionary Access Control (DAC). Only when DAC

would allow a particular behaviour is the AppArmor policy consulted. [73], [74]

If we compare our analysis with the available Signal profile (relevant part of the profile below) [75], we can see that we covered the same ground as this profile. For example, the `Signal` directory in the configuration directory, access to various font configuration caches, and that most of the behaviour is the same as any other Electron application.

```
@{name} = signal-desktop{,-beta}
@{lib_dirs} = @{lib}/signal-desktop "/opt/Signal{, Beta}"
@{config_dirs} = "@{user_config_dirs}/Signal{, Beta}"
@{cache_dirs} = @{user_cache_dirs}/@{name}

@{exec_path} = @{lib_dirs}/@{name}
profile signal-desktop @{exec_path} {
  include <abstractions/base>
  include <abstractions/audio-client>
  include <abstractions/common/electron>
  include <abstractions/fontconfig-cache-read>
  include <abstractions/user-download-strict>

  network inet dgram,
  network inet6 dgram,
  network inet stream,
  network inet6 stream,
  network netlink raw,
```

Looking at the Telegram profile at AppArmor.d website, it looks like we covered the necessary file path in our strace analysis. However, perhaps by mistake, the AppArmor.d project also includes `electron` directories in the `telegram-desktop` profile, which allows access to paths it does not need to see (Telegram uses Qt, not Electron).

We can see that AppArmor is great for restricting file access if the profile is well configured. Using strace is also a good way to get the information needed to create such profile.

AppArmor is good for restricting access to services such the internet access. However, once it allows the access to a service, it has much less control – if the internet access is allowed, it can send and receive packets from any website. Therefore, even with a tight AppArmor profile, analysis of the internet traffic is necessary.

4.3 Wireshark

During our analysis we also ran `netstac -nc`, however, we focused more on the analysis with Wireshark.

We looked at the statistical summaries provided by Wireshark. The Packet Length and Protocol Hierarchy summaries seemed to provide useful data for our analysis, but we mainly relied on the Endpoints TCP and UDP statistics. We manually looked up information for each IP address mentioned, as there were not too many of these different IPs, and we grouped them by the entity associated with a given IP. Finally, we put all the information into a short summary table.

In Signal, we tested text messages and phone calls.

In both tables we excluded private networks, multicast, unicast and limited broadcast IP addresses, so there is a difference between the totals and the numbers in the tables.

In the case of a voice call via signal-desktop, most of the communication was via IPv6, 51 MiB - 86 678 packets via UDP and 3 MiB - 7 870 packets via TCP. The rest of the information can be found in Table 1. We can see Amazon in the table because Signal hosts its servers mainly on Amazon Web Services (AWS). Orange is the Internet Service Provider (ISP) at the location where we tested the connection, and R-KOM Regensburger Telekommunikationsverwaltungsgesellschaft should be an ISP in Bavaria, Germany. The connection to Manjaro is just background noise related to our operating system. We can see a very similar result in Table 2 only with unsurprisingly much lower data usage for text messaging with Signal.

The main disadvantage of Wireshark is that we are unable to easily associate the connection with processes, so beside the traffic of analysed application, we can see background noise

Entity	Sum - Packets	Sum - MiB
Amazon.com Inc.	2218	1.1080
Cloudflare, Inc.	821	0.1648
Google LLC	1924	0.2583
Hetzner Online GmbH	30	0.0037
Manjaro	31	0.0032
Orange Slovensko, a. s.	47237	26.8888
R-KOM Regensburger	43137	25.4710
Total Result	97000	54.1500

Table 1: Wireshark TCP Endpoints of signal-desktop call

Entity	IPV 4/6	UDP/TCP	Sum - MiB
Amazon.com Inc.	4	TCP	0.0004
	6	TCP	0.1046
CLOUDFLARENET	6	TCP	0.0183
Hetzner Online GmbH	6	TCP	0.0012
Manjaro	4	TCP	0.0010
Orange Slovensko, a. s.	6	TCP	0.1242
		UDP	0.0035
Total Result			0.2548

Table 2: Wireshark TCP Endpoints of signal-desktop text messaging

such as automatic checking for updates. Also, most of the traffic is encrypted, so we are unable to inspect the content of most of communication with Wireshark, as we do not have the decryption keys.

4.4 BCC tool

After the preliminary analysis with `strace` and Wireshark, we tried to reproduce this analysis with faster tools that only trace the data we really need. To make sure that this method does not generate false positives, we tried it by comparing the official `signal-desktop` with the unofficial but quite reputable `signal-cli`.

In order to do this, we needed to use a suitable tracing tool. We described the options available in Chapter 2 and decided to use eBPF with the help of the BCC toolkit. BCC performs majority of its tracing in-kernel, which significantly speeds up tracing, which is the main reason why we chose it.

We mainly used the tcpconnect, opensnoop and execsnoop commands from the BCC repository. For additional analysis of file manipulation, filegone can also be used. We used tcpconnect to trace active connections via `connect()` system calls. opensnoop to monitor opened files via `open()` system calls, and execsnoop for monitoring new processes via `exec()`.

4.4.1 connect

According to the signal-desktop discussion on the SignalComunity forum and the `dns.ts` file found on Signal's Github, the current production domain names that Signal should connect to are `chat.signal.org`, `storage.signal.org`, `cdsi.signal.org`, `cdn.signal.org`, `cdn2.signal.org` and `create.signal.art` and using TCP port 443 and all UDP ports. [76], [77]

So, if Signal applications connect to some other domains, this could indicate a suspicious activity.

Below we can see the log of the `connect()` system call, where `PID` is the process ID of the application making the connection, `COMM` is the command or application name making the network request, `DADDR` is the destination address, i.e. the IP address of the server, `DPORT` is the destination port, commonly 443 for HTTPS connections and 80 for HTTP, `QUERY` is the domain name being queried.

The original log also includes `IP` - the IP protocol version used for the connection (6 for IPv6, 4 for IPv4) and `SADDR` source address, i.e. the IP address of the local machine which we omitted for better readability.

In Table 3 we can see that the official `signal-desktop` only makes the expected connections. The `ping.manjaro.org` is just a standard background noise that occurs even when no application is running and is not related to the Signal application itself but to the operating system.

Apart from the fact that the `signal-cli` uses IPv4 addresses instead of IPv6, the domain names it connects to remain the same. An interesting fact shown in Table 4 is that sending

PID	COMM	DADDR	DPORT	QUERY
13646	signal-deskt	2600:9000:a61f:527c: d5eb:a431:5239:3232	443	chat.signal.org
13646	signal-deskt	2606:4700:4400::ac40:966c	443	cdn2.signal.org
13646	signal-deskt	2606:4700:4400::ac40:966c	443	cdn2.signal.org
13646	signal-deskt	2606:4700:4400::ac40:966c	443	cdn2.signal.org
13646	signal-deskt	2a00:1450:4014:80e::2013	443	storage.signal.org
13646	signal-deskt	2600:9000:a61f:527c: d5eb:a431:5239:3232	443	chat.signal.org
13646	signal-deskt	2600:9000:2611:9200: 1d:4f32:50c0:93a1	443	cdn.signal.org
13646	signal-deskt	2600:9000:a61f:527c: d5eb:a431:5239:3232	443	chat.signal.org
448	NetworkManag	2a01:4f8:c0c:51f3::1	80	ping.manjaro.org
448	NetworkManag	116.203.91.91	80	ping.manjaro.org

Table 3: Network connections for signal-desktop

PID	COMM	DADDR	DPORT	QUERY
11234	signal-cli	13.248.212.111	443	chat.signal.org
11234	.signal.org/	13.248.212.111	443	chat.signal.org

Table 4: Network connections for signal-cli – send a message

requires only two connections to `chat.signal.org` while linking your desktop device with your main device, receiving all messages or just messages from a single user requires the same number of connections.

4.4.2 open

After comparison, we found that opensnoop gives us comparable information to strace, without the need to filter out data.

PID	COMM	DADDR	DPORT	QUERY
11084	signal-cli	13.248.212.111	443	chat.signal.org
11084	.signal.org/	13.248.212.111	443	chat.signal.org
11084	.signal.org/	13.248.212.111	443	chat.signal.org
11084	tokio-runtim	2603:1030:7::1	443	cdsi.signal.org
11084	signal-cli	104.18.37.148	443	cdn2.signal.org
11084	pool-6-threa	3.161.119.11	443	cdn.signal.org
11084	pool-6-threa	142.251.36.147	443	storage.signal.org

Table 5: Network connections for signal-cli – receive all messages

PID	COMM	DADDR	DPORT	QUERY
448	NetworkManag	116.203.91.91	80	ping.manjaro.org
448	NetworkManag	2a01:4f8:c0c:51f3::1	80	ping.manjaro.org
11413	signal-cli	76.223.92.165	443	chat.signal.org
11413	.signal.org/	76.223.92.165	443	chat.signal.org
11413	.signal.org/	76.223.92.165	443	chat.signal.org
11413	pool-6-threa	142.251.36.147	443	storage.signal.org

Table 6: Network connections for signal-cli – receive messages from a single contact

PID	COMM	DADDR	DPORT	QUERY
10958	.signal.org/	13.248.212.111	443	chat.signal.org
448	NetworkManag	116.203.91.91	80	ping.manjaro.org
448	NetworkManag	2a01:4f8:c0c:51f3::1	80	ping.manjaro.org
10958	signal-cli	76.223.92.165	443	chat.signal.org
10958	signal-cli	76.223.92.165	443	chat.signal.org
10958	signal-cli	76.223.92.165	443	chat.signal.org
10958	.signal.org/	76.223.92.165	443	chat.signal.org
10958	signal-cli	3.161.119.11	443	cdn.signal.org
10958	pool-9-threa	142.251.36.147	443	storage.signal.org

Table 7: Network connections for signal-cli – link

We also tracked the `open()` system call to monitor what files it tries to access. Both signal-desktop and `signal-cli` are expected to access mainly their own configuration files and various libraries, as well as the contents of `/usr/share/` and `~/.local/share`.

The `signal-desktop` tries to access files mostly connected to form its own files and configuration files `/usr/lib/signal-desktop/`, `~/.config/Signal`, but most of the directories are connected to GUI visuals such as fonts and icons `/usr/share/fonts`, `~/.local/share/fonts/`, `~/.fontconfig`, `/.cache/fontconfig/`, `~/.icons/`, `~/.local/share/icons`. Then there are directories such as `/usr/share/locale/`, various libraries from `/usr/lib/`. The log of system call also shows that the application uses the Chromium based Electron framework as it uses `/dev/shm/.org.chromium.Chromium.*` and `/tmp/.org.chromium.Chromium.*` and in this version also the Vulkan 3D graphics API `~/.local/share/vulkan/` and `/usr/share/vulkan/` API and open standard for 3D graphics and computing. It also accesses `/etc/passwd`, which could indicate an enumeration attempt if it was malicious software, but given that this is the official

application and no other signs of enumeration have been detected, we can consider it legitimate.

Once again, signal-cli proves to be a legitimate application that is much easier to monitor than signal-desktop, mainly because it has no GUI and therefore produces less noise.

In the same way as signal-desktop, it accesses `/etc/passwd`, `/dev/urandom`, `/etc/ld.so.cach`, `/etc/nsswitch.conf`, `/etc/resolv.conf`, `/sys/devices/system/cpu/possible`, `/proc/stat`, `/proc/self/maps`, `/proc/self/fd` `/run/systemd/machines/chat.signal.org`.

From the general directories we listed as being accessed by `signal-desktop`, only `/usr/lib/` was accessed by `signal-cli`. Therefore, we can assume that the rest of the directories accessed by signal-desktop were required by the Electron framework GUI and were not needed for the core Signal functionality. The `signal-cli` also accesses its own local directory `~/.local/share/signal-cli` and instead of the Chromium temporary directory it creates a shared object file (.so) related to the SQLite JDBC driver inside the temporary directory, which is probably used for Java database connectivity with SQLite. It also tries to ensure data integrity by creating a corresponding lock file to prevent multiple processes from accessing this shared object.

Overall, the activity of `signal-cli` seems to be legitimate, it mostly accesses the same files as `signal-desktop`, and even thanks to being simple command-line software, it accesses much fewer files than `signal-desktop`.

4.4.3 exec

The final part of the analysis is to find out if signal-desktop and signal-cli execute additional software only when they really need to. Furthermore, this analysis shows us in detail what arguments are used to start these commands.

In Table 8 we can see which programs are executed by signal-desktop. In `signal-desktop` (PIDs 13546, 13549, 13550) Signal's desktop application launches and child processes for zygote are created, where a zygote process is the one that listens for spawn re-

quests from the main process and forks itself in response. The fact that it disables `--no-zygote-sandbox` reduces security by allowing broader access to system resources, but it is not seriously concerning. The flag `xdg-settings` (PIDs 13566, 13582) sets the default URL scheme handlers for Signal. Manipulating URL handlers can be risky if misused by other applications, but in this context, it appears to be a legitimate configuration for Signal. Finally, the `exe` – `/proc/self/exe` (PIDs 13616, 13646, 13679) represent various utility types being run (network service, renderer, and audio service), likely from within Signal. Here again there are flags for disabling of features like `HardwareMediaKeyHandling` and `SpareRendererForSitePerProcess`, and flags such as `--no-sandbox` and `--enable-crash-reporter` reduce security. For example, disabling the sandbox might be exploited via the rendering engine or enabling crash reports might leak sensitive data. [78]

Table 8: exec system calls of signal-desktop

PID	COMM	PPID	RET	ARGS
13541	fish	1	0	"/usr/bin/fish"
13545	flatpak	13541	0	"/usr/bin/flatpak" "–installations"
13546	signal-desktop	13541	0	"/usr/bin/signal-desktop"
13549	signal-desktop	13546	0	"/usr/lib/signal-desktop/signal-desktop" "–type=zygote" "–no-zygote-sandbox"
13550	signal-desktop	13546	0	"/usr/lib/signal-desktop/signal-desktop" "–type=zygote"
13566	xdg-settings	13546	0	"/usr/bin/xdg-settings" "set" "default-url-scheme-handler" "sgnl" "signal.desktop"
13582	xdg-settings	13546	0	"/usr/bin/xdg-settings" "set" "default-url-scheme-handler" "signalcaptcha" "signal.desktop"

Table 8 continued from previous page

PID	COMM	PPID	RET	ARGS
13616	exe	13546	0	"/proc/self/exe" "–type=utility" "–utility-sub-type=network.mojom.NetworkService" ... "–service-sandbox-type=none" ... "--disable-features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess" "–variations-seed-version"
13646	exe	13546	0	"/proc/self/exe" "–type=renderer" ... "–no-sandbox" "–no-zygote" "–enable-blink-features=CSSPseudoDir,CSSLogical" "--disable-blink-features=Accelerated2dCanvas,AcceleratedSmallCanvases" "–first-renderer-process" "–disable-gpu-compositing" ... "–enable-features=kWebSQLAccess" ""
13679	exe	13546	0	"/proc/self/exe" "–type=utility" "–utility-sub-type=audio.mojom.AudioService" "–lang=en-US" "–service-sandbox-type=none" ...

Table 9: exec system calls for signal-cli send

PCOMM	PID	PPID	RET	ARGS
signal-cli	11234	10307	0	"./signal-cli -u +421909104930 send -m "Hello, how are you?" +421909543173"
sh	11237	11234	0	"/usr/bin/sh -c stty -a < /dev/tty"
stty	11239	11237	0	"/usr/bin/stty -a"
sh	11240	11234	0	"/usr/bin/sh -c stty -a < /dev/tty"
stty	11241	11240	0	"/usr/bin/stty -a"
uname	11242	11234	0	"/usr/bin/uname -o"

As we can see in Table 9, `signal-cli` performs fewer and much less complex steps than

`signal-desktop`, the same as with the `connect()` and `open()` system calls. It initiates a shell `sh` (PIDs 11237, 11240) to execute further commands. Then it executes `stty` (PIDs 11239, 11241), `stty -a` to be precise, which gets all the settings for the current terminal. It uses `uname` (PID 11242), `uname -o`, which gets the name of the operating system. As all actions end here, we can assume that these operations are done just to use the correct settings for the given operating system. The same commands are executed for linking devices and receiving messages.

4.5 Analysis of repacked snap application

Snap is a framework for distributing desktop applications in the form of application packages across different Linux distributions. Snap typically runs in a restricted and transaction-based environment. [79] Its main competitor and alternative is Flatpak.

When we rerun our analysis done in Section 4.2 but with the snap version of the application, the results are almost the same, just with small changes caused by the restricted environment of snap. These changes are mainly that some files and directories are in `/home/jackie/snap/` instead of their previous location, but the rest of their file path remains the same.

We were provided with the repacked version of `signal-desktop` snap application. In addition to the standard behaviour of the application, it also connects to the external server where this communication takes place.

```
GET / HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: tQGjEGDkY08boTjyz/3FNw==
Host: attacker.example.com:13579
Sec-WebSocket-Protocol: echo-protocol

HTTP/1.1 101 Switching Protocols
```

```
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 3Zzizf3O1jqlVxsCORsi9Kv1gUM=
Sec-WebSocket-Protocol: echo-protocol

..Abrakadabra
```

Mirroring Section 4.4, where we ensured that our analysis did not produce false positives. Here we tried to provide proof of concept that our analysis does not produce false negatives and that we can detect this simulated attack.

The analysis went as follows. We ran all our tracing tools: strace, Wireshark as well as BCC tools. Then, in Wireshark we could see a new IP address that was not there before, as well as DNS communication for a completely new domain, outside of what was expected. We could also see similar results in tcpconnect. We went through the output of the strace and BCC tools opensnoop, execsnoop, filegone to check for any signs of suspicious behaviour.

Conclusion

This thesis focused on the behaviour of Signal and Telegram related to system interactions captured through system calls while performing basic operations like sending and receiving messages.

We identified the behaviour patterns of applications Signal and Telegram. By comparing the official desktop application, the GUI framework on which the instant messaging application is based, and the minimal TUI version of the given application, we were able to identify the core behaviour of Telegram and Signal.

We attempted a thorough review of the observability tools, explored their mechanisms in order to evaluate the strengths as well as the drawbacks of each tool. We reached the conclusion that the eBPF – the in-kernel extension of the Berkeley Packet Filter, originally used only by `tcpdump` for packet filtering, now used in many ways from sandboxing in browsers to tracing of almost any part of the OS – is the best choice. Not only because it allows us to use a single tool to do all the tracing, but also because it does all its tracing in the kernel and sends the final results to the user-space, which speeds up the tracing significantly and makes running of the software while tracing it virtually as fast as running it without tracing, unlike tools like `strace` which can sometimes slow down programs to the point of failure. Since we decided to use eBPF, we used the BCC toolkit for BPF-based Linux IO analysis, networking and monitoring.

First, we performed preliminary analysis using more established tools such as strace and Wireshark. After that we tried the same with eBPF.

We explored the differences between the official Signal application, which operates on the Electron framework incorporating the Chromium browser engine, and its unofficial command-line counterpart, signal-cli, which utilises a custom patched version of libsignal-service from the Signal-Android source code. The official signal-desktop version tested was 7.0.0-1, sourced from the stable Manjaro repository, while signal-cli was evaluated at version 0.13.3 from the official developer's repository. The analysis involved tracing system interactions through system calls and network logs. Signal-desktop displayed more

complex, interconnected operations due to its GUI-based framework, whereas signal-cli facilitated more straightforward, discrete command-line operations, allowing for more granular analysis. Both applications were confirmed to communicate only with official Signal service endpoints over secure connections. Furthermore, they allowed us to identify the core behaviour of the Signal application that is shared by both.

We then repeated this procedure with Telegram. We compared telegram-desktop, version 5.1.5-1 sourced from the stable Manjaro repository, with a Qt example application, as Telegram uses the Qt framework. This allowed us to identify the core pattern of Telegram.

In conclusion, the analysis of Signal and Telegram applications demonstrates the effectiveness of leveraging system call monitoring and unique file path identification to uncover application-specific behaviour, even within the applications with encrypted communication.

The Signal case study highlighted the potential of this strategy to reveal detailed behaviour patterns from encrypted network traffic without prior knowledge of the communication protocol or security architecture.

Our attempts with strace and BCC showed that system call monitoring could facilitate the identification of Signal's chat, voice, and video call events, as well as disclose the IP addresses and servers involved in these communications. This approach serves as a proof of concept for observation of application behaviour, but also emphasises the importance of continuous monitoring and updates, particularly for applications built on complex and frequently updated frameworks like Electron.

Our analysis of Telegram also shows that the same approach is well suited to other instant messaging applications that use different GUI frameworks and network infrastructures.

Future research could extend these methodologies to other social media applications and adapt the monitoring strategy to different IP ranges and ports, thereby broadening the scope and applicability of this analysis technique.

References

[1] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE symposium on security and privacy*, 2012, pp. 95–109. doi: 10.1109/SP.2012.16[1]

[2] H. Rafiq, N. Aslam, M. Aleem, B. Issac, and R. H. Randhawa, "AndroMalPack: Enhancing the ML-based malware classification by detection and removal of repacked apps for android systems," *Scientific Reports*, vol. 12, no. 1, p. 19534, Nov. 2022, doi: 10.1038/s41598-022-23766-w[2]. Available: https://doi.org/10.1038/s41598-022-23766-w

[3] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi, "AdRob: Examining the landscape and impact of android application plagiarism," in *Proceeding of the 11th annual international conference on mobile systems, applications, and services*, in MobiSys '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 431–444. doi: 10.1145/2462456.2464461[3]. Available: https://doi.org/10.1145/2462456.2464461

[4] L. Štefanko and P. Strýček, "Not-so-private messaging: Trojanized WhatsApp and Telegram apps go after cryptocurrency wallets." https://www.welivesecurity.com/2023/03/16/not-so-private-messaging-trojanized-whatsapp-telegram-cryptocurrency-wallets/, Mar. 16, 2023.

[5] Android Open Source Project, "Privacy changes in Android 10." https://developer.android.com/about/versions/10/privacy/changes#clipboard-data, 2023.

[6] D. Kalinin, "Analysis of a spy module inside a WhatsApp mod." https://securelist.com/spyware-whatsapp-mod/110984/, Nov. 02, 2023.

[7] L. Štefanko, "StrongPity espionage campaign targeting Android users." https://www.welivesecurity.com/2023/01/10/strongpity-espionage-campaign-targeting-android-users/, Jan. 10, 2023.

[8] L. Štefanko, "BadBazaar espionage tool targets Android users via trojanized Signal and Telegram apps." https://www.welivesecurity.com/en/eset-research/badbazaar-espionage-tool-targets-android-users-trojanized-signal-telegram-apps/, Aug. 30, 2023.

[9] I. Golovin, "Spyware Telegram mod distributed via Google Play — securelist.com." https://securelist.com/trojanized-telegram-mod-attacking-chinese-users/110482/, Sep. 08, 2023.

[10] A. Hart, Leptopoda, and Y. Lu, "What's with the protocol using Rust?" https://community.signalusers.org/t/whats-with-the-protocol-using-rust/18957, Dec. 20, 2020.

[11] B. Gregg, "BSidesSF 2017: Security monitoring with eBPF." https://www.brendangregg.com/Slides/BSidesSF2017_BPF_security_monitoring, 2017.

[12] B. Gregg, "What is Observability." https://brendangregg.com/blog/2021-05-23/what-is-observability.html, May 23, 2021.

[13] J. Evans, "Linux tracing systems & how they fit together." https://jvns.ca/blog/2017/07/05/linux-tracing-systems/, 2017.

[14] B. Gregg, "Choosing a Linux Tracer (2015)." https://brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html, 2015.

[15] J. Edge, "Unifying kernel tracing." https://lwn.net/Articles/803347/, 2019.

[16] Terenceli, "Linux tracing - kprobe, uprobe and tracepoint." https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2020/08/05/tracing-basic, 2020.

[17] P. Kranenburg, "Strace - an alternative syscall tracer, Part01/04." https://stuff.mit.edu/afs/sipb/project/eichin/cruft/machine/sun/sun-Strace, 1992.

[18] B. Gregg, "Strace Wow Much Syscall." https://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html, May 11, 2014.

[19] D. Levin, "FOSDEM 2020 - Postmodern strace." https://archive.fosdem.org/2020/schedule/event/debugging_strace_modern/, Feb. 02, 2020.

[20] E. Syromyatnikov, "FOSDEM 2020 - strace: Fight for performance." https://archive.fosdem.org/2020/schedule/event/debugging_strace_perfotmance/, Feb. 02, 2020.

[21] A. Morton, "Re: Linux-next: Add utrace tree." https://lwn.net/Articles/371505/, 2010.

[22] J. Corbet, "Replacing ptrace()." https://lwn.net/Articles/371501/, 2010.

[23] T. Heo, "Proposal for ptrace improvements." https://lwn.net/Articles/430373/, 2011.

[24] J. Corbet, "Improving ptrace()." https://lwn.net/Articles/432114/, 2011.

[25] M. Müller, "Can a process have multiple parents? Why or why not?" https://stackoverflow.com/questions/42333659/can-a-process-have-multiple-parents-why-or-why-not, 2017.

[26] Andrew, "Bypassing ptrace in gdb." https://stackoverflow.com/questions/33646927/bypassing-ptrace-in-gdb, 2015.

[27] OffSecServices, "OffSec's Exploit Database Archive — exploit-db.com." https://www.exploit-db.com/exploits/40839, 2016.

[28] Y. Shafet, "The Race to Limit Ptrace." https://www.rezilion.com/blog/the-race-to-limit-ptrace/, 2020.

[29] J. Evans, "A zine about strace." https://jvns.ca/blog/2015/04/14/strace-zine/, 2015.

[30] S. Abdalla, "Say this five times fast: Strace, ptrace, dtrace, dtruss." https://dev.to/captainsafia/say-this-five-times-fast-strace-ptrace-dtrace-dtruss-3e1b, 2019.

[31] N. Abda, "Understanding ptrace — abda.nl." https://abda.nl/posts/understanding-ptrace/, 2019.

[32] S. McCanne, G. Combs, and L. Degioanni, "SHARKFEST '11 Keynote Address." https://sharkfestus.wireshark.org/sharkfest.11/, 2011.

[33] tshepang, "Difference between sniffer tools — networkengineering.stackexchange.com." https://networkengineering.stackexchange.com/a/10075, 2017.

[34] wireshark, "Tools - Wireshark Wiki — wiki.wireshark.org." https://wiki.wireshark.org/Tools, 2020.

[35] user862787, "Performance and efficiency comparing between dump tools: Tcpdump, tshark, dumpcap." https://stackoverflow.com/a/22234865, 2014.

[36] J. Evans, "Let's learn tcpdump!" https://wizardzines.com/zines/tcpdump/, 2017.

[37] eBPF Foundation, "eBPF Projects." https://ebpf.foundation/projects/.

[38] M. Steve, "Shark fest 2011 keynote." https://sharkfestus.wireshark.org/sharkfest.11/, 2011.

[39] S. McCann, "The BSD packet filter: A new architecture for user-level packet capture." http://www.tcpdump.org/papers/bpf-usenix93.pdf, 1992.

[40] L. Rice, "Learning eBPF — chapter 1." https://www.oreilly.com/library/view/learning-ebpf/9781098135119/ch01.html, 2023.

[41] Linux Kernel, "BPF licensing." https://www.kernel.org/doc/html/latest/bpf/bpf_licensing.html.

[42] A. Starovoitov, "BPF in-kernel virtual machine." https://netdevconf.info/0.1/sessions/15.html, 2015.

[43] E. Dumazet, "Re: [PATCH v2] net: Filter: Just in time compiler." Email to David Miller, Apr. 03, 2011. Available: https://lore.kernel.org/netdev/1301838968.2837.200.camel@edumazet-laptop/

[44] Chromium Project, "The seccomp sandbox." Online. Available: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/linux/sandboxing.md#The-sandbox-1

[45] A. Chapman, "Seccomp and Seccomp-BPF." https://ajxchapman.github.io/linux/2016/08/31/seccomp-and-seccomp-bpf.html, 2016.

[46] Kernel Documentation, "Seccomp BPF (SECure COMPuting with filters)." Online. Available: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html

[47] Linux Kernel Developers, "Source code of xt_bpf.c in linux kernel v3.9." Online, 2013. Available: https://elixir.bootlin.com/linux/v3.9/source/net/netfilter/xt_bpf.c

[48] Linux Kernel Developers, "Source code of cls_bpf.c in linux kernel v3.13." Online, 2014. Available: https://elixir.bootlin.com/linux/v3.13/source/net/sched/cls_bpf.c

[49] Kernel Documentation, "Linux socket filtering aka berkeley packet filter (BPF)." Online, 2024. Available: https://www.kernel.org/doc/html/latest/networking/filter.html

[50] D. Borkmann, "Re: [PATCH net-next v4 0/9] BPF updates." Email to David Miller, Mar. 28, 2014. Available: https://lore.kernel.org/netdev/1396029506-16776-1-git-send-email-dborkman@redhat.com/

[51] Wikipedia contributors, "History of eBPF." Wikipedia, The Free Encyclopedia, May 2024. Available: https://en.wikipedia.org/wiki/EBPF#History

[52] Polynka, "Lua in the kernel?" https://lwn.net/Articles/831083/, 2020.

[53] A. Dinaburg, "Pitfalls of relying on eBPF for security monitoring (and some solutions)." https://blog.trailofbits.com/2023/09/25/pitfalls-of-relying-on-ebpf-for-security-monitoring-and-some-solutions/, 2023.

[54] B. Gregg, A. Robertson, *et al.*, "Bpftrace internals." Online, 2024. Available: https://github.com/bpftrace/bpftrace/blob/master/docs/internals_development.md

[55] B. Gregg *et al.*, "BPF compiler collection." Online, 2024. Available: https://github.com/iovisor/bcc/blob/master/README.md

[56] B. Gregg, "eBPF observability tools are not security tools." https://brendangregg.com/blog/2023-04-28/ebpf-security-issues.html, 2023.

[57] J. Gregorio, A. Gardel, and B. Alarcos, "Forensic analysis of telegram messenger for windows phone," *Digital Investigation*, vol. 22, pp. 88–106, 2017, doi: https://doi.org/10.1016/j.diin.2017.07.004. Available: https://www.sciencedirect.com/science/article/pii/S1742287617301032

[58] K. Rathi, U. Karabiyik, T. Aderibigbe, and H. Chi, "Forensic analysis of encrypted instant messaging applications on android," in *2018 6th international symposium on digital forensic and security (ISDFS)*, 2018, pp. 1–6. doi: 10.1109/ISDFS.2018.8355344[4]

[59] A. Afzal, M. Hussain, S. Saleem, M. K. Shahzad, A. T. S. Ho, and K.-H. Jung, "Encrypted network traffic analysis of secure instant messaging application: A case study of signal messenger app," *Applied Sciences*, vol. 11, no. 17, 2021, doi: 10.3390/app11177789[5]. Available: https://www.mdpi.com/2076-3417/11/17/7789

[60] K. Gupta, D. Oladimeji, C. Varol, A. Rasheed, and N. Shahshidhar, "A comprehensive survey on artifact recovery from social media platforms: Approaches and future research directions," *Information*, vol. 14, no. 12, 2023, doi: 10.3390/info14120629[6]. Available: https://www.mdpi.com/2078-2489/14/12/629

[61] M. N. Yusoff, A. Dehghantanha, and R. Mahmod, "Chapter 5 - network traffic forensics on firefox mobile OS: Facebook, twitter, and telegram as case studies," in *Contemporary digital forensic investigations of cloud and mobile applications*, K.-K. R. Choo and A. Dehghantanha, Eds., Syngress, 2017, pp. 63–78. doi: https://doi.org/10.1016/B978-0-12-805303-4.00005-8. Available: https://www.sciencedirect.com/science/article/pii/B9780128053034000058

[62] "Windows 10 Mobile End of Support: FAQ." Microsoft, 2019. Available: https://support.microsoft.com/en-us/windows/windows-10-mobile-end-of-support-faq-8c2dd1cf-a571-00f0-0881-bb83926d05c5

[63] C. Reilly, "Goodbye, Windows 10 Mobile, tweets Joe Belfiore — cnet.com." https://www.cnet.com/tech/mobile/windows-10-mobile-features-hardware-death-sentence-microsoft/, 2017.

[64] C. Anglano, M. Canonico, and M. Guazzone, "Forensic analysis of telegram messenger on android smartphones," *Digital Investigation*, vol. 23, pp. 31–49, 2017, doi: https://doi.org/10.1016/j.diin.2017.09.002. Available: https://www.sciencedirect.com/science/article/pii/S1742287617301767

[65] S. Scheibner, https://github.com/AsamK/signal-cli/issues/1522#issuecomment-2103284397, 2024.

[66] Electron, "Electron-quick-start-typescript." https://github.com/electron/electron-%20quick-start-typescript, 2024.

[67] Qt, "All Qt Examples | Qt 6.9 — doc-snapshots.qt.io." https://doc-snapshots.qt.io/qt6-dev/qtexamples.html, 2024.

[68] kpcyrd, "Arch Linux - signal-desktop 7.8.0-1 (x86_64) — archlinux.org." https://archlinux.org/packages/extra/x86_64/signal-desktop/, 2024.

[69] Signal, "Signalapp/Signal-Desktop." https://github.com/signalapp/Signal-Desktop, 2024.

[70] Signal, "Signal-Android/libsignal-service." https://github.com/signalapp/Signal-Android/tree/main/libsignal-service, 2023.

[71] S. Scheibner, "Release v0.13.3 · AsamK/signal-cli." https://github.com/AsamK/signal-cli/releases/tag/v0.13.3, 2024.

[72] A. Pujol, "GitHub - roddhjav/apparmor.d: Full set of AppArmor profiles." https://github.com/roddhjav/apparmor.d, 2024.

[73] AppArmor, "GettingStarted · Wiki · AppArmor." https://gitlab.com/apparmor/apparmor/-/wikis/GettingStarted, 2023.

[74] Ubuntu, "Ubuntu server documentation - AppArmor." https://ubuntu.com/server/docs/apparmor, 2024.

[75] A. Pujol *et al.*, "Signal-desktop profile at roddhjav/apparmor.d." https://github.com/roddhjav/apparmor.d/blob/main/apparmor.d/groups/apps/signal-desktop, 2024.

[76] Herohtar, "URLs for Prioritizing on my firewall." https://community.signalusers.org/t/urls-for-prioritizing-on-my-firewall/38553/4, 2023.

[77] Signal, "Firewall and Internet settings." https://support.signal.org/hc/en-us/articles/360007320291-Firewall-and-Internet-settings.

[78] Chromium, "Chromium Docs - docs/linux/zygote.md." https://chromium.googlesource.com/chromium/src/+/HEAD/docs/linux/zygote.md.

[79] Snap, "Snap documentation." https://snapcraft.io/docs, 2024.

[80] OpenAI, "ChatGPT - GPT-4o (May 16, 2024 version). In the preparation of this thesis, this AI tool was used for summary, code and command assistance purposes. Any direct output has been thoroughly revised and rewritten to the point where there is unlikely to be any trace of generated output. The author is responsible for the accuracy of the final text." https://chat.openai.com, 2024.