Progger 3: A Low-Overhead, Tamper-Proof Provenance System

A thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Master of Cyber Security

at

The University of Waikato

by

Tristan Corrick

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2021

Abstract

Data provenance, which describes how data is accessed and used since the time it is created, is a valuable resource with a wide range of uses. It can be used simply to know who has accessed one's data, or be used in more complex scenarios such as detecting malware. One method for collecting data provenance is to observe system calls. This thesis presents Progger 3, a system that observes system calls on Linux in order to collect data provenance. There are several existing provenance systems that observe system calls, but they have limitations regarding security, efficiency, and usability. Progger 3 remedies many of these limitations. As a result, Progger 3 is a working implementation of a provenance system that can observe any system call, guarantee tamper-proof provenance collection as long as the kernel on the client is not compromised, and transfer the provenance to other systems with confidentiality and integrity, all with a relatively low performance overhead.

Acknowledgements

I would like to thank my supervisor, Dr Vimal Kumar, for his guidance and support with my work. I would also like to thank the University of Waikato and the STRATUS project for providing a Research and Enterprise Study Award, which provided financial assistance with this research.

Contents

1 Introduction **2**

1.1 Design goals of Progger 3 4

1.2 Thesis outline 5

2 Background **6**

2.1 System calls 6

2.2 Kernel space and user space 7

2.3 TPMs 7

2.4 Trusted kernels 8

2.5 XChaCha20-Poly1305 9

3 Existing Provenance Systems **11**

3.1 Progger 1 12

3.1.1 Kernel-only implementation in Progger 1 13

3.2 Progger 1 trusted framework 14

3.3 Progger 2 14

3.3.1 Kernel-only implementation in Progger 2 15

3.4 PASS 16

3.5 CamFlow 17

3.6 Sysdig 17

3.7 SystemTap 18

3.8 bpftrace 18

3.9 Comparison to Progger 3 19

4 The Architecture of Progger 3 **21**

4.1 Chapter outline 22

4.2 An overview of Progger 3 22

4.3 Record format 24

4.3.1 Header format 25

4.3.2 Body format 25

4.3.3 Server JSON output 28

4.4 Kernel-only operation 29

4.4.1 Trusted kernels . . . 29
4.4.2 Kernel-only implementation in Progger 3 . . . 30
4.4.3 Conclusion . . . 31
4.5 Cryptography . . . 32
4.5.1 Confidentiality . . . 33
4.5.2 Integrity . . . 33
4.5.3 Cryptography approach in Progger 3 . . . 33
4.5.3.1 Private key storage . . . 36
4.6 Trusted platform module . . . 38
4.6.1 TPM provisioning . . . 39
4.6.2 TPM unsealing . . . 40
4.6.3 TPM benefits . . . 41
4.7 Performance improvements . . . 41
4.8 Ability to trace any system call . . . 42
4.9 Stability . . . 44
4.10 Maintainability . . . 45
4.11 Conclusion . . . 46

5 TPM Usage Proof of Correctness 47

6 Evaluation 51
6.1 Impact on system throughput . . . 52
6.1.1 Impact on CPU throughput . . . 53
6.1.1.1 Time to compile Linux 5.8.3 . . . 53
6.1.1.2 Time to compile Linux 5.8.3 with no traced system calls occurring . . . 58
6.1.1.3 Impact on a CPU-bound program making no system calls . . . 60
6.1.2 Impact on network throughput . . . 60
6.2 Impact on system latency . . . 61
6.2.1 Impact on a system call being traced . . . 62
6.2.2 Impact on a system call not being traced . . . 67
6.3 Correctness . . . 69
6.4 Summary . . . 69

7 Discussion 71
7.1 Chapter outline . . . 71
7.2 Comparison with similar work . . . 71
7.2.1 Progger 1 and a TPM . . . 71
7.2.2 Detecting commands executed as a different user . . . 73
7.3 Future work . . . 76

7.3.1 Utilising the disk to store buffers 77
7.3.2 User space network interface control 77
7.3.3 Task priority 78
7.3.4 Copying system call pointer arguments 79
7.3.5 Reducing bandwidth usage 80
7.3.6 Information leakage 80
7.3.7 Namespaces 80
7.3.8 The Progger 3 server 81
7.3.9 Choosing which system calls to trace 81
7.4 Summary 82

8 Conclusion **83**

References **84**

Appendices **88**

A Progger 3 source code **89**
A.1 Kconfig (diff) 89
A.2 drivers/net/Makefile (diff) 89
A.3 drivers/net/progger/Kconfig 90
A.4 drivers/net/progger/Makefile 90
A.5 drivers/net/progger/kernel/Kconfig 90
A.6 drivers/net/progger/kernel/Makefile 93
A.7 drivers/net/progger/kernel/crypto.c 94
A.8 drivers/net/progger/kernel/crypto.h 95
A.9 drivers/net/progger/kernel/generated/gen-ip.py 96
A.10 drivers/net/progger/kernel/generated/gen-syscalls.py 96
A.11 drivers/net/progger/kernel/init.c 98
A.12 drivers/net/progger/kernel/kthread.c 99
A.13 drivers/net/progger/kernel/kthread.h 100
A.14 drivers/net/progger/kernel/net.c 100
A.15 drivers/net/progger/kernel/net.h 104
A.16 drivers/net/progger/kernel/ringbuf.c 104
A.17 drivers/net/progger/kernel/ringbuf.h 107
A.18 drivers/net/progger/kernel/tpm.c 108
A.19 drivers/net/progger/kernel/tpm.h 116
A.20 drivers/net/progger/kernel/tracepoints.c 116
A.21 drivers/net/progger/kernel/tracepoints.h 124
A.22 drivers/net/progger/scripts/tpm/provision 124
A.23 drivers/net/progger/scripts/tpm/provision-inner 125

A.24 drivers/net/progger/scripts/tpm/provision-setup 129
A.25 drivers/net/progger/server/Makefile 129
A.26 drivers/net/progger/server/crypto.c 130
A.27 drivers/net/progger/server/crypto.h 131
A.28 drivers/net/progger/server/server.c 131
A.29 drivers/net/progger/server/syscall-table.c 140
A.30 drivers/net/progger/server/syscalls.h 146
A.31 include/progger/compiler.h 147
A.32 include/progger/crypto.h 147
A.33 include/progger/net.h 148
A.34 include/progger/record.h 148
A.35 include/progger/types.h 149

B The `new-session` program **151**

List of Figures

2.1 TPM architecture overview [5] 8

4.1 Progger 3 configuration with `make nconfig` 23

4.2 Progger 3 configuration: tracepoints sub-menu 24

4.3 The record header format 25

4.4 The `RECORD_SYSCALL_X86_64` format 26

4.5 Format of messages sent over the network by Progger 3 34

6.1 Linux compile test results 56

6.2 Linux compile test results 59

6.3 Time taken for `openat` to complete 63

6.4 Median time taken for `openat` to complete 65

6.5 99th percentile of the time taken for `openat` to complete . . . 66

6.6 Median time taken for `openat` to complete while not being traced 67

6.7 99th percentile of the time taken for `openat` to complete while not being traced 68

List of Tables

3.1 Comparison of Progger 3's design goals with other provenance systems, to the best of our understanding 20

6.1 Real time to compile Linux 57

6.2 Real time to compile Linux, no traced system call occurring . 59

Chapter 1

Introduction

For a set of data, its provenance is the metadata that is required to answer certain questions about the history of that set of data [1]. These questions may include: "Where did this set of data originate?", "What transformations has this set of data undergone over time?", "Who has used this set of data?", and "How was this set of data obtained?" [1]. This thesis uses "provenance" on its own to mean "data provenance". No other types of provenance are discussed.

Provenance is a powerful tool that has the potential to solve many problems. For example, in an experiment it can provide a link between final results and initial parameters, even when the data passes through many complex stages [1], aiding in scientific reproducibility. Furthermore, if one separates results into expected and unexpected, comparing the differences in provenance between the two groups can be used to help identify the cause of the unexpected results [1]. This is especially useful for software debugging, and also allows retroactive debugging. To give a final example: provenance can assist with system intrusion detection, by monitoring for abnormalities in the provenance that is being generated.

The exact metadata constituting provenance is not fixed. Instead, there are multiple approaches to collecting provenance. One might choose to trace C library calls an application makes, for example, or instead choose to trace system calls. The approach of tracing system calls to collect provenance is

what the focus of this thesis is.

To give a more concrete example of tracing system calls to collect provenance, consider the provenance consisting of records of all `open`, `openat`, and `openat2` system calls executed by Linux on a given system. These records could contain the system call arguments, such as file paths, the time the system call occurred, the user making the system call, as well as whether the operation was successful. This provenance, in particular, could be useful for monitoring system intrusion.

Provenance is collected by *provenance systems*. This thesis presents Progger 3: a low-overhead, tamper-proof provenance system. Progger 3 traces Linux system calls in order to collect provenance. As its name suggests, there have been two earlier iterations of Progger on Linux.

The first iteration was Progger 1. Progger 1's name is really just "Progger", but we have retroactively named it Progger 1 to distinguish it from later iterations. Progger 1 was created by Ryan K. L. Ko and Mark A. Will [2].

The second iteration, Progger 2 was developed at the University of Waikato, but has not seen a public release. It began to use tracepoints to observe system calls, rather than rewriting the addresses of system call functions as done by Progger 1.

Given these two iterations, there were still problems that could be resolved. Primarily: performance. When using Progger 1 or Progger 2, our tests showed a considerable reduction in system performance. Additionally, Progger 2 showed instability in our tests, causing user space programs to experience segmentation faults, and causing system crashes. Both Progger 1 and Progger 2 have only a limited set of system calls that can be traced. Also, they have user space components or bugs that mean they are not kernel-only provenance systems, and hence prone to a malicious user space tampering with provenance collection to some extent. Neither offers confidentiality and complete integrity of data as it is transferred over the network. There is a proposed framework for Progger 1 that provides this transit confidentiality and integrity [3], but

it requires an existing provenance client implementation, and it is also only intended for a few specific use cases.

Progger 3 was created to remedy these issues. Progger 3 was designed to be truly kernel-only so that a malicious user space cannot tamper with the provenance at any stage. Progger 3 combines its kernel-only mode with a Trusted Platform Module (TPM), which allows it to extend the tamper-proof property to the provenance as it is sent over the network to a remote server. This is because combining a TPM with the kernel-only mode secures a cryptographic key from user space. This has two effects. First, the provenance can be transferred over the network encrypted, providing confidentiality and integrity, meaning attackers on the network path cannot tamper with or read the provenance. Secondly, as user space can never access the cryptographic keys, it cannot generate false provenance records. Furthermore, Progger 3 was designed with efficiency as a primary objective, and as such is realistically usable with many workloads without causing an unacceptable drop in performance. It also has the ability to trace any system call, greatly enriching the provenance that can be collected. We believe that Progger 3 is the first iteration of Progger that can realistically be used under a wide range of workloads.

1.1 Design goals of Progger 3

The following are the design goals of Progger 3. They are presented early, as later sections will be clearer with these in mind. It is very important to keep in mind that Progger 3 is designed to prevent an untrusted user space from being able to maliciously impact Progger 3's operation (in kernel space), as some design decisions might seem counterproductive otherwise.

A The provenance system is kernel-*only*, meaning that user space cannot alter the provenance system client's code, configuration (except during compilation), or any data generated by the provenance system client, both at rest and at runtime.

B User space is never able to generate false provenance that would go undetected when received by the server.

C The provenance has confidentiality in transit.

D The provenance has integrity in transit.

E Collecting and transferring the provenance has a minimal performance impact.

F Any system call can be traced.

G Provenance collection can begin before user space starts.

H The provenance system cannot be unloaded once loaded.

I The provenance system is stable; that is, crashes are rare.

J Existing APIs are used to trace system calls.

1.2 Thesis outline

Chapter 2 briefly explores some concepts that are fundamental to the understanding of later chapters. Chapter 3 investigates some existing provenance systems and distinguishes them from Progger 3. Chapter 4 details the architecture of Progger 3 and explains how the architecture allowed Progger 3's design goals to be achieved. Chapter 5 proves the correctness of Progger 3's TPM usage. Chapter 6 evaluates the performance of Progger 3, as well as some other relevant provenance systems. Chapter 7 discusses several aspects of Progger 3, such as future work and similarities with other research. Chapter 8 summarises the thesis. Finally, in Appendix A, the source code of the implementation of Progger 3 can be found.

Chapter 2

Background

To aid with understanding the later chapters, this chapter will discuss some key background topics relating to Progger 3. These topics are: system calls, kernel space and user space, TPMs, trusted kernels, and XChaCha20-Poly1305.

2.1 System calls

System calls are functions that an operating system kernel provides so that user space can request that the kernel perform certain operations. These exist because the kernel is in charge of managing many aspects of the system, such as file systems in many cases, so user space has to make requests to the kernel if it wants to use the file systems. For example, Linux provides system calls such as `open` to open a file on a file system, given a file path, and `read` / `write` to modify that file.

The use of system calls in Linux is particularly relevant, as Progger 3 is a Linux kernel module. There are over 300 system calls for x86-64, as of Linux 5.8 [4]. These range from performing file accesses, to creating new processes, to networking [4].

Given how extensively system calls are required on Linux for accessing and modifying data, observing the system calls and their arguments provides an avenue for data provenance collection with high levels of comprehensiveness.

2.2 Kernel space and user space

When code executes on a system, it is generally in either kernel space or user space. As the names suggest, kernel space is where code from the operating system kernel executes, and user space is where code run by the users of the system executes. These two spaces are isolated from each other, as kernel space code has higher privileges than user space. This isolation is achieved through several measures, such as having separate address spaces, so user space cannot access the memory of kernel space. That is, unless the kernel opts to offer this access to user space. This isolation is an important security measure, as it means that errant user space programs cannot crash the whole system by overwriting important structures in kernel memory. By making sure the kernel doesn't provide user space with any access that would subvert this separation, it is possible to consider the code running in kernel space trusted while considering the code in user space untrusted. This idea of a trusted kernel and untrusted user space is fundamental to Progger 3.

2.3 TPMs

A Trusted Platform Module (TPM) is "a system component that has state that is separate from the system on which it reports" [5]. In more concrete terms, it is a device that provides cryptographic services to a system, such as storing the state of the system as SHA256 hashes and encrypting data. Such a device is generally useful when one has some level of distrust in the code running on the system. This is because, by the TPM storing the state of the system as a cryptographically secure hash, it is possible to determine if the system is in a state that is considered good, and take actions dependent on that. For example, as a system boots, it can verify each successive component in the boot sequence, and represent this as a hash stored in one of the many platform configuration registers (PCRs). Then, one can inspect these PCRs to determine whether the system has booted using only trusted components.

TPMs have a wide range of uses, and as such are quite complex. In terms of understanding their use with Progger 3, it is generally enough to know that TPMs can reflect system state, as previously described, and seal data so that the TPM must be used to decrypt it. Furthermore, the TPM can enforce that decryption is only allowed when a PCR equals a particular value. This is the mechanism that Progger 3 will use to secure cryptographic keys. An overview of the components in a TPM can be seen in Figure 2.1.

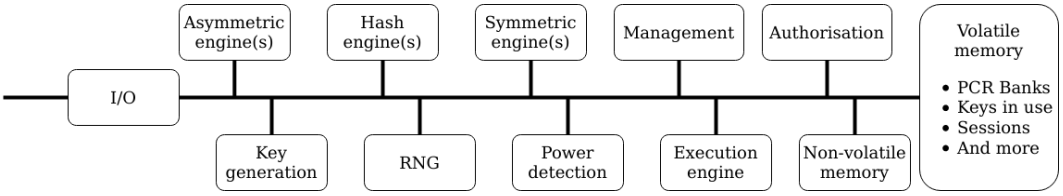

Figure 2.1: TPM architecture overview [5]

2.4 Trusted kernels

There are many ways to achieve trust in the kernel on a running system. A comprehensive explanation of how to achieve this trust is outside the scope of Progger 3, but an overview is still relevant.

The first requirement to achieve trust in the kernel is that each component in the boot process must verify the next. For a typical x86-64 system, starting from the reset vector, the host firmware executes, such as BIOS/UEFI/coreboot. The host firmware will perform its tasks, then verify the bootloader, such as GRUB, usually through checksumming, and pass control to the bootloader if verification succeeds. Then, the bootloader is able to verify the kernel in a similar manner and boot it.

Perhaps the most natural question to ask, when hearing about each component verifying the next in the boot flow, is: what verifies the host firmware? This can be done in many ways. One approach is to use a system that supports

Intel Boot Guard [6]. Using Boot Guard means that any firmware image not signed by the system manufacturer will be rejected [7]. Alternatively, the host firmware could write-protect the flash chip it resides upon, enforcing it either through the platform controller hub [8] or the flash chip itself [9]. Flash chip write-protection is different from verifying the host firmware signature at each boot; rather, it ensures that the host firmware cannot be modified through software. Only users with physical access to the system and the appropriate hardware to write to the flash chips can modify the firmware. This approach is preferable to some, as it means the owner has more control over their system.

Secondly, Linux must boot with `lockdown=confidentiality` set in the kernel command line. For the `lockdown` parameter, specifying `integrity` prevents user space from being able to alter the running kernel. Specifying `confidentiality` maintains the restrictions from `integrity` while also preventing user space from extracting secrets from the kernel [10].

With the above recommendations in place, the kernel can be considered secure and trusted in case of a user space compromise. This is, of course, barring bugs in the hardware, kernel, host firmware, and bootloader that might allow an attacker to boot unsigned kernels, or alter the running kernel. It is worth noting that there are other domains of execution on modern systems that could alter the kernel at runtime. For an x86-64 system, System Management Mode (SMM) is one such domain. Defending against attacks in these domains is outside the scope of Progger 3, so we will leave the discussion at that.

With this knowledge, it should be possible to create a system with a trusted kernel that can take advantage of Progger 3's kernel-only mode.

2.5 XChaCha20-Poly1305

XChaCha20-Poly1305 is a cryptographic algorithm used in Progger 3 to provide confidentiality and integrity of provenance in transit. To understand **X**ChaCha20-Poly1305, this section first explores ChaCha20-Poly1305.

ChaCha20-Poly1305 is an Authenticated Encryption with Associated Data (AEAD) algorithm [11]. It takes as input a 256-bit key K, a 96-bit nonce n (in the IETF specification of the algorithm), a plaintext of arbitrary length, and additional authenticated data (AAD) of arbitrary length [11]. The algorithm produces a ciphertext that is the same length as the plaintext, and a 128-bit tag t that is used for authentication [11]. To be clear, the plaintext is encrypted, but the AAD is not. When decryption occurs, the input is K, t, n, the ciphertext, and the AAD. This produces the plaintext, but only if the ciphertext *and* AAD are successfully authenticated against t [11]. This means that a ciphertext will fail to authenticate if a different set of AAD is present, even if that AAD is valid with some other ciphertext encrypted with K.

XChaCha20-Poly1305 is essentially the same algorithm as ChaCha20-Poly1305, but the nonce is 192-bits in XChaCha20-Poly1305 [12]. The larger nonce is the reason that Progger 3 uses XChaCha20-Poly1305, and the reason that the larger nonce is needed is discussed in subsection 4.5.3.

Chapter 3

Existing Provenance Systems

A large number of provenance systems exist today. Pérez *et al.* have recently systematically reviewed provenance systems and identified 251 provenance systems in total [13]. Provenance systems have a wide range of scope, from database provenance, to specific scientific workflows, to operating system events [1]. Progger 3 is based on operating system events, system calls in particular. So, to keep the overview of provenance systems in this chapter relevant, this chapter only covers provenance systems that trace operating system events.

Provenance systems that are based on operating system events are important for several reasons. Their use means that user space programs do not have to be modified for collection of provenance and provides a wide overview of activity on a system. It is also possible for a trust boundary between the operating system kernel and user space can be established, as described in section 2.4. With a provenance system operating in kernel space, collecting data based on operating system events, one can have high assurances that the provenance collected is accurate (in particular, has not been falsified).

This chapter explores existing provenance systems, and briefly analyses how they differ from Progger 3. The exploration focuses mostly on the limitations of these provenance systems; however, this is because we wish to show where Progger 3 has made improvements. These systems certainly have their

own merits, but it is not always as relevant to detail them here. The improvements Progger 3 has made over some of these existing provenance systems are described in more detail throughout chapter 4.

At the end of this chapter, Table 3.1 summarises the existing provenance systems explored in this chapter in terms of Progger 3's design goals.

3.1 Progger 1

The first provenance system this chapter will look at is Progger 1. As previously mentioned, Progger 1 was created by Ryan K. L. Ko and Mark A. Will [2].

Progger 1 collects provenance by monitoring a predefined set of system calls, including, but not limited to, `open`, `read`, `write`, `sendmsg`. The log format varies depending on the particular system call. The exact formats can be found in [2], but it largely consists of the system call arguments, as well as the user and process IDs associated with the current running task. The log format is text-based, which leads to a slight decrease in the efficiency of data transfer compared to a binary format. These logs are written to the kernel log buffer with `printk`, and `rsyslog` takes care of transferring the logs to remote systems.

In order to observe the system calls, Progger 1 rewrites the address of the system call functions to ones defined by Progger 1, which wrap the original system calls. This is not ideal, as the system call table is marked read-only, and hardware write protection (bit 16 of register CR0) is temporarily disabled for a short time to rewrite the function addresses. This gives a small window of time where read-only data may be updated, whether maliciously or not, which could harm the system.

One of Progger 1's significant breakthroughs was providing a level of tamper-*evident* logging, so that falsified provenance could be detected in many (but not all) cases [2]. While the provenance collected by Progger 1 has some integrity,

there is no confidentiality when it is in transit [2].

Being a nascent development, Progger 1 had not been optimised much for performance, so it can increase the execution time of the `open` system call by up to 21,000% [2].

One notable issue is that Progger 1 is available *only* as a standalone module: there is no option for it to be built into the kernel. So, there will be a gap where provenance is not collected before user space loads the module.

3.1.1 Kernel-only implementation in Progger 1

This section will see how Progger 1 fares in terms of Progger 3's design goal **A**. Progger 1 is a standalone Linux kernel module, but there are some aspects that are dependent on user space programs. One such program is `rsyslog`, used for transferring collected provenance [2]. There are two classes of issue here: the availability of the provenance, and the tamper-resistance of the provenance. Some issues are:

1. The `rsyslog` process could be killed or stopped, preventing the provenance from being logged.

2. A compromised root user could alter `rsyslog` so that it doesn't transfer certain provenance records, or adds false provenance records.

3. An arbitrary kernel module could generate false provenance records.

These issues are similar to, or were exactly specified, in [2]. The recommended approach in [2] places trust in the `rsyslog` process by passing the process ID of `rsyslog` to the kernel module as it is inserted. Unfortunately, this means that, if `rsyslog` is killed, it will almost certainly respawn with a different PID, so the provenance gathering would halt.

An approach presented by [2] to solve the issue of false records being added was a hash-chaining approach. They came to the conclusion that such an approach makes log tampering difficult, but not impossible [2].

So, Progger 1 is not a kernel-only implementation, and some issues arise as a result. Yet, Progger 1 was never developed with an entirely kernel-only mode as an objective [2].

3.2 Progger 1 trusted framework

M. M. M. Bany Taha explored creating a trusted framework built upon Progger 1 [3]. It has some similar goals to Progger 3; namely, providing confidentiality and integrity for the collected provenance. However, it goes beyond the scope of Progger 3 to present a framework that encompasses collecting provenance from clients, securely storing the provenance in servers, then later archiving the provenance. This framework uses software around an existing provenance client in order to make improvements. Progger 3, meanwhile, addresses existing shortcomings by improving the provenance client itself. Due to this approach, some issues in Progger 1, such as efficiency, are not dealt with by this framework.

This framework is compared with Progger 3 in more detail in subsection 7.2.1, which shows more clearly why there is still a need for Progger 3 given the existence of this extension to Progger 1.

3.3 Progger 2

Progger 2 was developed at the University of Waikato, but has not seen a public release. It began to use tracepoints in order to observe system calls, rather than rewriting the addresses of system call functions as done by Progger 1. Like Progger 1, it supports only a fixed subset of the available system calls. It opted to create a direct TCP connection to a server for sending provenance, rather than the `rsyslog`-based approach in Progger 1, removing the need for user space components during run-time. As a whole, Progger 2 was, unfortunately, quite unstable in our tests, causing system crashes when used. From the brief evaluation in chapter 6, the performance hit was too high for many

workloads. For the provenance logged, there was no guarantee of integrity or confidentiality. These issues can mostly be explained by the fact that Progger 2 had not finished development to at least a release-quality level.

As with Progger 1, Progger 2 is available only as a standalone kernel module, so cannot capture provenance early in the boot process.

3.3.1 Kernel-only implementation in Progger 2

As done for Progger 1, this section evaluates how closely Progger 2 meets Progger 3's design goal **A**. Progger 2 is a standalone Linux kernel module. In contrast to Progger 1, it doesn't have any user space components.

There is at least one subtle but serious flaw in Progger 2: it uses `sock_create` instead of `sock_create_kern` (which is what Progger 3 uses) to create the socket used for sending the provenance over TCP. Following the code in Linux, one arrives at the code found in Listing 3.1.

Listing 3.1: Linux's `inet6_create` function

```
1  static int inet6_create(struct net *net, struct socket *sock,
2                          int protocol, int kern)
3  {
4      [...]
5      if (!kern) {
6          err = BPF_CGROUP_RUN_PROG_INET_SOCK(sk);
7          if (err) {
8              sk_common_release(sk);
9              goto out;
10         }
11     }
12 out:
13     return err;
14     [...]
```

This means that, as the socket is being created, an eBPF program of the type `BPF_PROG_TYPE_CGROUP_SOCK` can be run. This eBPF program, created by user space, can determine whether to allow or block the creation of the socket [14]. The code example above is specifically for an IPv6 socket, but this holds true for IPv4 sockets too. Hence, user space is able to block Progger 2 from starting. Of course, it would be simple to replace the call to `sock_create`

with `sock_create_kern`, but this does highlight how easy it is to make subtle mistakes.

Although not strictly related to being kernel-only, it is an opportune moment to point out that, since Progger 2 does not authenticate what it sends, a malicious user space process could start generating false provenance. It could be possible to detect this: consider receiving two streams of provenance at once. However, combined with user space's ability to deny Progger 2 from starting, a malicious process could quite realistically masquerade as the legitimate Progger 2 process.

So, Progger 2 strived to be a kernel-only implementation, but there are, unfortunately, some issues that stop it from getting there completely.

3.4 PASS

PASS [15] is a provenance system with a slightly larger scope than Progger 3: it seeks to collect provenance through observing system calls, managing the storage of the provenance, and allow queries to be performed on the provenance [15, 16]. For storage, local file systems, network file systems, or cloud storage can all be used [16].

PASS is a Linux kernel module, but to our knowledge has not described steps to ensure that user space is unable to impact the operation of PASS maliciously. Indeed, if transferring the provenance to the cloud, it is described that a file system in user space (FUSE), named PA-S3fs, is used [16]. Further, the authors of PASS state that work on provenance tamper-proofing is complimentary to their work [16].

Additionally, although it is up to the user of PASS to configure their remote file systems and cloud storage correctly, it would be possible to ensure the integrity and confidentiality of the provenance in transit if the selected storage mechanism supports it.

3.5 CamFlow

CamFlow [17] is a provenance system that has both user space and kernel space components. Some of its authors were involved in the development of PASS [17].

In kernel space, CamFlow has a Linux Security Module (LSM), as opposed to a traditional Linux kernel module like each Progger iteration. An LSM is able to provide code to run at specific security-critical points in the kernel [18]. The potential advantages of an LSM-based approach are briefly discussed in subsection 7.3.9.

CamFlow doesn't have the ability to trace only a select few system calls. In fact, system calls are not its primary target: its LSM-based approach centres around determining what objects are being accessed, rather than how the objects are being accessed [17]. Still, it can collect provenance based on several filters, such as provenance of specific files or users [17]. Whether this approach is more useful than Progger 3's depends on the particular user's needs.

As CamFlow has user space components, it is not a kernel-only provenance system. Of course, with different threat models, CamFlow's architecture can be perfectly suitable; but it doesn't fill the niche that Progger 3 does.

Unfortunately, we experienced a kernel panic and a user space utility segmentation fault after only a short time of testing, so stability has room for improvement.

3.6 Sysdig

Sysdig [19] is not typically described as a provenance system, but it has the capabilities to collect the same information from system calls as Progger 3. It acts similarly to the Progger iterations: it has a Linux kernel module that obtains provenance from system calls using tracepoints. It then writes that provenance to a user space buffer, where user space utilities manage reading it. It does not indicate a goal of tamper-proof provenance, and certainly, by design

does not prevent user space attacks. This is not surprising, as it presents itself as more of a debug and system observation tool, not a security tool.

3.7 SystemTap

SystemTap [20] is very similar to Sysdig. All the information given in the above section about Sysdig applies to SystemTap, except that SystemTap uses kprobes as well as tracepoints. The main difference is in their features, implementation, and performance. What is relevant here is their difference in performance, with chapter 6 showing that SystemTap performs better generally.

3.8 bpftrace

bpftrace [21] is again not typically described as a provenance system, but can be used as one, much like Sysdig and SystemTap. The main difference from the other provenance systems explored in this chapter is that bpftrace uses eBPF. (Sysdig can use eBPF, but its support is not entirely stable.) eBPF is a way of running sandboxed programs within the Linux kernel, allowing access to kernel programming interfaces while minimising the risk of errant programs damaging the kernel [22]. With eBPF, both tracepoints, kprobes, and more can be used to observe system calls. bpftrace makes it easy to compile an eBPF program and get the program's output into user space. The eBPF programs are loaded into the kernel from user space, so naturally the provenance is not protected from user space attacks. Again, this is not a security tool so much as it is a debugging tool that happens to be able to function as a provenance system.

3.9 Comparison to Progger 3

The design goals of Progger 3 are listed in section 1.1. As this thesis will later show, they have all been met.

Table 3.1 shows how the provenance systems from this chapter fare in terms of Progger 3's design goals. None of the other provenance systems comes close to meeting all of the design goals for Progger 3, and the difference is rather marked when comparing Progger 3 to Progger 1 and Progger 2. As such, we believe that the development of Progger 3 is justified and a useful contribution.

Progger 3 was tested on only two systems during development: one bare-metal system, and one virtual machine. This may mask stability issues with Progger 3, as any issues arising from system-specific quirks would have been fixed during development. It is possible that, on a different system, Progger 3 could exhibit reduced stability. Still, the experience of developing Progger 3 did not suggest much room for system-specific quirks to lead to reduced stability. Hence, it is likely that Progger 3 is widely stable.

	Design goal of Progger 3 (as given in section 1.1)									
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
Progger 1	×	partial	×	partial	×	×	×	× [a]	?	×
Progger 2	×	×	×	×	×	×	×	×	×	✓
Progger 3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PASS	×	×	✓	✓	borderline [15]	?	✓	?	?	?
CamFlow	×	×	N/A	N/A	borderline [17]	N/A	✓	× [b]	×	✓
Sysdig	×	×	N/A	N/A	borderline	✓	×	×	✓	✓
SystemTap	×	×	N/A	N/A	✓	✓	×	×	✓	✓
bpftrace	×	×	N/A	N/A	borderline	✓	×	×	✓	✓

Table 3.1: Comparison of Progger 3's design goals with other provenance systems, to the best of our understanding

[a] The code indicates an attempt at preventing module unloading, but doing so hangs the system.

[b] The LSM component cannot be removed, but the user space components can be instructed to stop provenance collection.

Chapter 4

The Architecture of Progger 3

Section 1.1 presents the design goals of Progger 3. Progger 3 has achieved all of its design goals, and this chapter will describe how the architecture of Progger 3 lends itself to achieving those goals. These goals can be roughly grouped into the following:

Efficiency. Progger 3 has been designed with efficiency as a primary objective. As can be seen in chapter 6, provenance systems such as Progger 1 and Progger 2 have so much overhead that they reduce performance to levels that may not be realistically usable. Progger 3 has remedied this, and is efficient enough to be used in many real workloads.

Security. By using a TPM, as well as ensuring a true kernel-only mode of operation exists, Progger 3 can essentially guarantee tamper-proof provenance collection, even as the provenance is transferred over the network to a remote server (assuming the kernel is not compromised).

Usability. Progger 3 allows one to easily select which system calls to monitor, and supports tracing any system call. With Progger 1 and Progger 2, the selection of system calls being monitored is fixed, and only a relatively small subset of all system calls are supported. Additionally, Progger 3 is very stable, meaning that it can be deployed with confidence and relied upon to continuously collect provenance without crashing.

4.1 Chapter outline

Section 4.2 gives an overview of Progger 3, exploring what Progger 3 is capable of and how it can be configured. Section 4.3 details the data format that the Progger 3 client uses to send records of system calls to a server. Section 4.4 describes how Progger 3 ensures a kernel-only mode of operation to guard against malicious user space activity. Section 4.5 discusses the cryptography used in Progger 3 to ensure the provenance's confidentiality and integrity. Section 4.6 explores the way in which Progger 3 utilises a TPM to fully achieve its cryptography and kernel-only design goals. Section 4.7 lists the steps taken to achieve performance improvements. Section 4.8 describes how Progger 3 is able to trace any system call without needing individual functions to support each specific system call. Section 4.9 covers the level of stability shown by Progger 3. That is, how much one can rely on Progger 3 to operate without errors or crashes. Section 4.10 discusses how the design of Progger 3 allows for easy maintenance and easy modification of features. Finally, section 4.11 summarises Progger 3's architecture and what Progger 3 has achieved by meeting its design goals.

4.2 An overview of Progger 3

Progger 3 is a Linux kernel module that collects data provenance through monitoring system calls. It uses tracepoints [1], an API of Linux allowing code to execute on entry and exit of certain functions, in order to log information about each system call. Progger 3 can be configured easily through Linux's `kbuild` system, by running `make menuconfig` or any other configuration interface, an experience that will be familiar to many who have compiled Linux before. The configuration interface is presented in Figure 4.1, and the tracepoint sub-menu, which allows one to provide a regular expression to select which system calls are traced, is presented in Figure 4.2. Any system call can be traced.

[1] This achieves design goal **J**

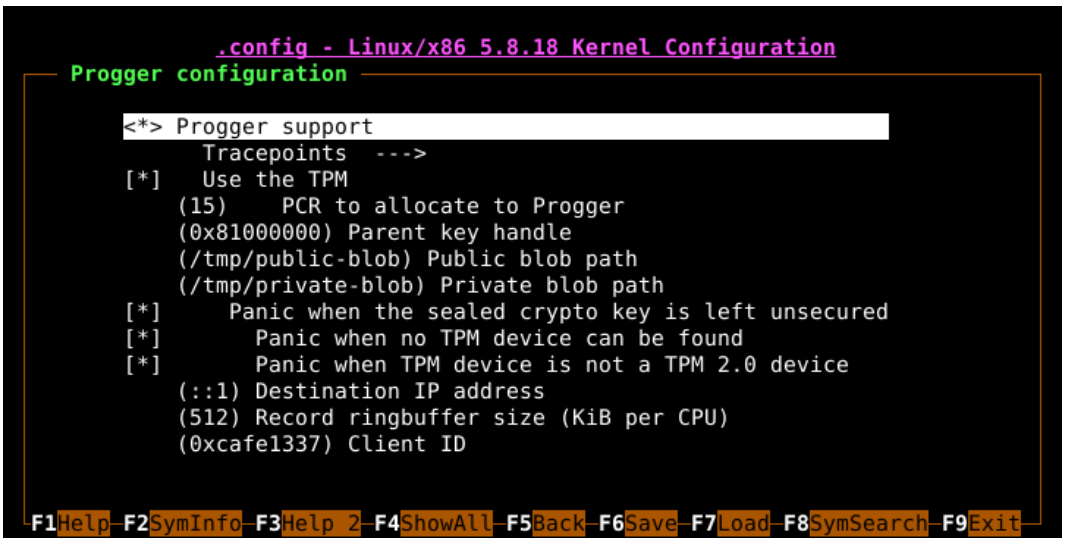

Figure 4.1: Progger 3 configuration with `make nconfig`

Values such as the system calls to trace and destination IP address have to be set at compile time, and cannot be changed during run time. This is because Progger 3 is designed so that user space cannot maliciously impact the operation of Progger 3, say by setting the list of traced system calls to be empty, or sending the (encrypted) provenance records to a different destination. This partially achieves design goal **A**.

Every time a system call that Progger 3 has been instructed to trace occurs, a provenance record is generated. There is currently no configuration to generate records for only a subset of the system calls that have been selected to be traced. For example, it is not possible to trace system calls made by only a specific process. As just mentioned, configuration must be made at compile time, not run time, and it is highly impractical to predetermine the PID of a particular process, except the init process. A process is able to change its own `comm` value, so that is not reliable either. Of course, process filtering can be done later by a program that processes the data collected by Progger 3.

While there are some limitations with flexibility, as just described, these limitations do exist for security purposes. In determining whether to use Progger 3, one should assess whether Progger 3's security advantages are beneficial

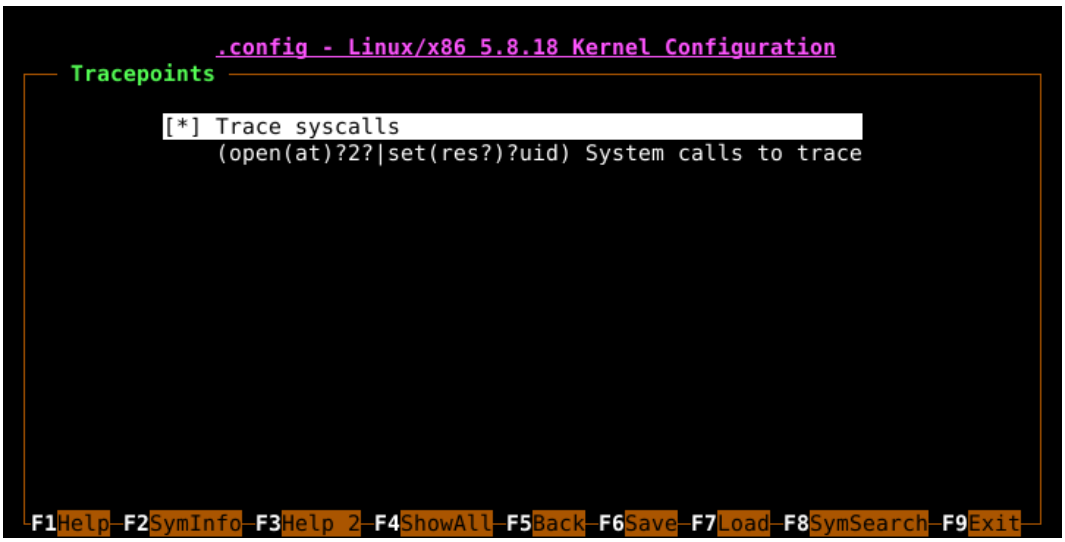

Figure 4.2: Progger 3 configuration: tracepoints sub-menu

under their threat model, and weigh those advantages up against Progger 3's more limited flexibility compared to other provenance systems.

When it comes to transferring the collected provenance, Progger 3 sends the data it collects over TCP to a server, which runs in user space. The server can be running on the same host as the Progger 3 client (for testing and debugging), or on a physically separate host that is reachable over the network. The remote server, collecting provenance from potentially multiple Progger 3 clients, is assumed to have a trusted user space. The implementation of the server can vary, but we have created an implementation that takes the data it receives and prints it as JSON. The next sections will define the record format Progger 3 uses to send data over the network, and then show the server's JSON output.

4.3 Record format

Each message Progger 3 sends across the network contains a set of *records* that are encrypted and authenticated by XChaCha20-Poly1305. A record is composed of two parts: a header and a body. The header allows for multiple record types in the future, but currently there is only one: `RECORD_SYSCALL_X86_64`.

4.3.1 Header format

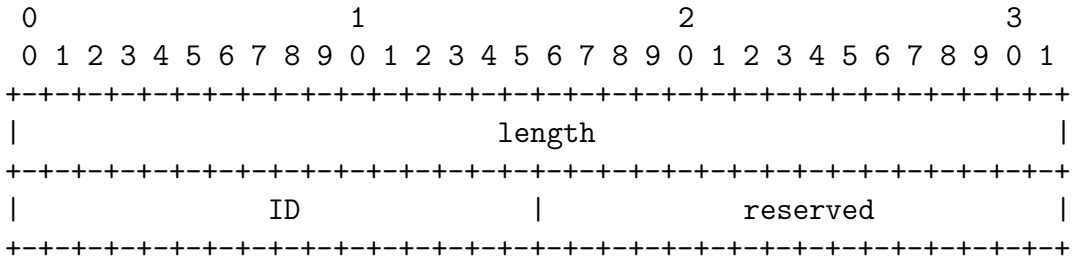

Figure 4.3: The record header format

The record header format is set out in Figure 4.3. Its fields have the following meanings:

- length — The length in bytes of the whole record, including the header. A 4-octet field.
- ID — The ID of the record. A 2-octet field.
- reserved — A reserved value to ensure the data following the header has a 4-octet alignment for performance. A 2-octet field.

4.3.2 Body format

Due to the record header just described, record types other than an x86-64 system call record can be easily added in the future. However, currently that is the only record that exists. Its record ID is `RECORD_SYSCALL_X86_64`. The format is set out in Figure 4.4. Each record represents one system call occurrence. Its fields have the following meanings:

- tracepoint src — The tracepoint that is the source of the data. MUST be either `TP_SRC_SYS_ENTER` (2) or `TP_SRC_SYS_EXIT` (4). Naturally, these values respectively refer to the `sys_enter` and `sys_exit` tracepoints. A 1-octet field.
- reserved — A reserved value so that successive fields have a 4-octet alignment. A 1-octet field.

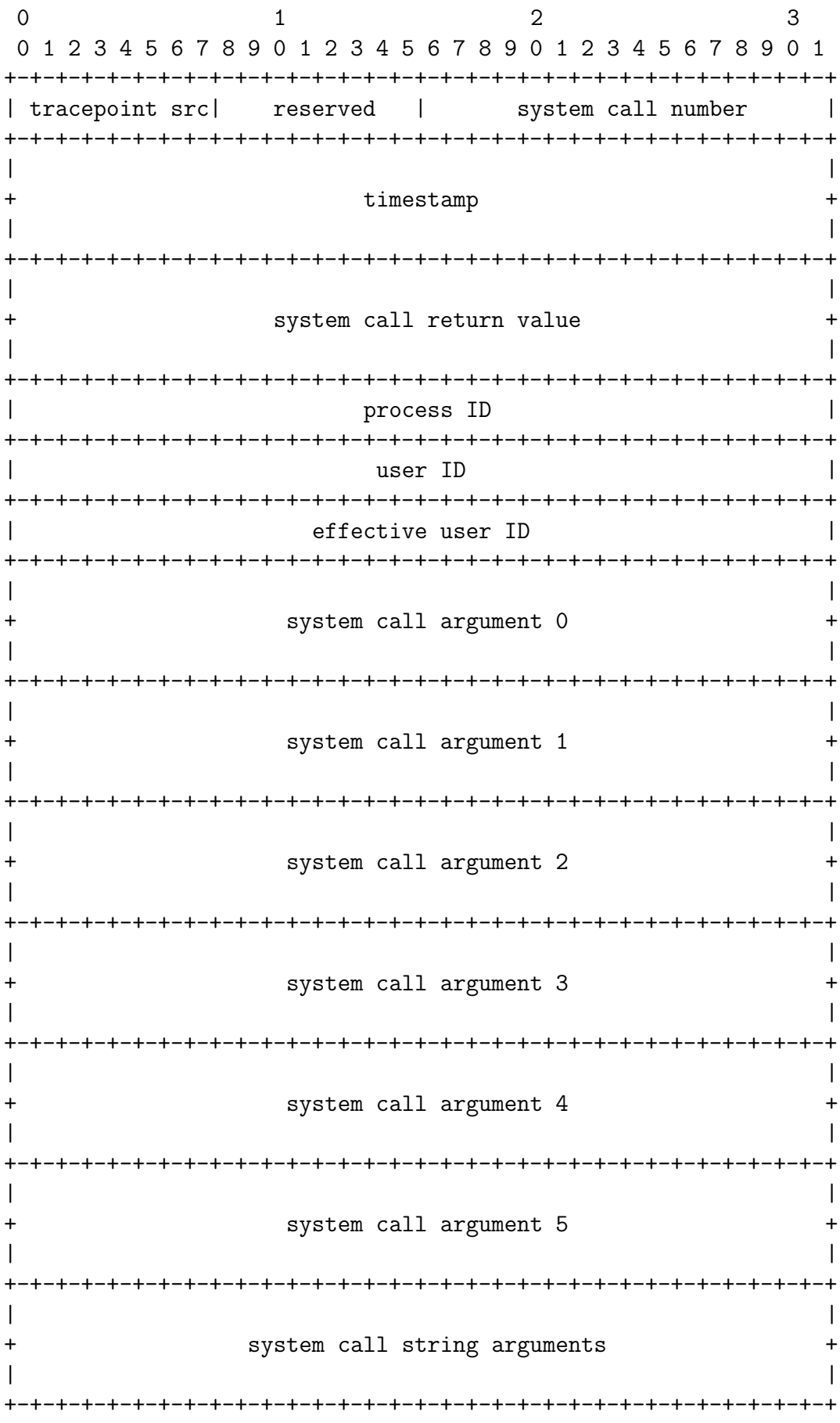

Figure 4.4: The RECORD_SYSCALL_X86_64 format

- system call number — The system call number, as given by Linux. A 2-octet field.

- timestamp — A timestamp representing the number of nanoseconds since the system booted. Some other time formats, such as the real value of the system clock, could be more desirable in some cases. However, the system clock can be modified by user space, so it is not suitable. Furthermore, the timestamp should be monotonic, which is not guaranteed of the system clock. A 4-octet field.

- system call return value — The return value of the system call, if the tracepoint source is `TP_SRC_SYS_EXIT`. 0 otherwise. A 4-octet field.

- process ID — The process ID of the process that made the system call, as seen from the initial PID namespace. A 4-octet field.

- user ID — The real user ID of the process that made the system call, as seen from the initial user namespace. A 4-octet field.

- effective user ID — The effective user ID of the process that made the system call, as seen from the initial user namespace. A 4-octet field.

- system call argument 0..5 — The arguments to the system call. If, for some n, the system call doesn't use argument n, the system call argument n field is undefined and should be ignored. Each is a 4-octet field.

- system call string arguments — If a system call takes a C-string for an argument, its value will appear here. The strings are concatenated from argument 0 to argument 5. Each string is limited to 4096 octets, including the NUL byte. Linux defines the maximum length of a file path to be 4096 bytes, also including the NUL byte, so string truncation should not lose any meaningful information. A variable-length field, up to 24576 octets.

A notable omission is that this record does not contain the contents pointed to by pointer arguments, other than C-strings. The approach taken by Progger 3, where only integer arguments and C-strings are copied, seeks to minimise complexity and maximise efficiency while still providing the most useful information for many use cases. Pointer arguments can point to complex `struct`s, which would take a lot of care to serialise and deserialise, increasing the risk for error. Errors are crucial to avoid in kernel code, as an error can lead to a system crash, or compromise at a very high privilege level.

4.3.3 Server JSON output

Listing 4.1 provides a sample of the output of our Progger 3 server implementation when the system calls `openat`, `setuid`, `setreuid`, and `setresuid` are being monitored. Refer to section 4.3 for more detail.

Listing 4.1: Progger 3 server output

```
1 { "id": "openat", "tp_src": "sys_exit", "ts": 8901772142832, "ret":
    6, "pid": 12300, "uid": 0, "euid": 0, "args": [ 4294967196,
    140735053210000, 591872, 0, 140735053211440, 32 ], "strings": [
    "\/etc\/gss\/mech.d" ] }
2 { "id": "setresuid", "tp_src": "sys_exit", "ts": 8901772503698,
    "ret": 0, "pid": 12300, "uid": 105, "euid": 105, "args": [ 105,
    105, 105, 7, 94176267341920, 94176267341824 ], "strings": [ ] }
3 { "id": "setuid", "tp_src": "sys_exit", "ts": 8901772511526, "ret":
    -1, "pid": 12300, "uid": 105, "euid": 105, "args": [ 0, 0, -1,
    7, 94176267341920, 94176267341824 ], "strings": [ ] }
4 { "id": "setresuid", "tp_src": "sys_exit", "ts": 8901772513575,
    "ret": -1, "pid": 12300, "uid": 105, "euid": 105, "args": [ -1,
    0, -1, 7, 94176267341920, 94176267341824 ], "strings": [ ] }
5 { "id": "openat", "tp_src": "sys_exit", "ts": 8901826793176, "ret":
    27, "pid": 217, "uid": 0, "euid": 0, "args": [ 4294967196,
    140736600249456, 524288, 0, 0, 1 ], "strings": [
    "\/proc\/12299\/comm" ] }
6 { "id": "openat", "tp_src": "sys_exit", "ts": 8901827120638, "ret":
    -2, "pid": 217, "uid": 0, "euid": 0, "args": [ 4294967196,
    140736600249184, 524288, 0, 0, 1 ], "strings": [
    "\/run\/systemd\/units\/log-extra-fields:ssh.service" ] }
```

4.4 Kernel-only operation

The Progger 3 client operates in kernel-only mode, which means that there are no user space components in the client, and that user space cannot alter the code of the client or any data produced by the client. Additionally, operating entirely in kernel mode means that data doesn't have to be copied between user space and kernel space, leading to efficiency gains. The following sections explore how Progger 3 achieves its kernel-only mode of operation; that is, achieves design goal **A**.

4.4.1 Trusted kernels

Having the Progger 3 client run entirely as a kernel module is necessary for a kernel-only mode of operation (design goal **A**), but it is not sufficient. To ensure kernel-only operation, the user must verify that the kernel itself cannot be tampered with; that is, that the kernel can be trusted. Achieving a trusted kernel is outlined in section 2.4. Since Progger 3 has a secret key in kernel memory, the user must boot with the kernel argument `lockdown=confidentiality`, as mentioned in section 2.4. Now, the kernel has dominion over user space, and user space is no longer able to modify the trusted kernel, assuming no bugs compromise this separation. Yet, these steps are still not entirely sufficient to achieve design goal **A**. The final steps taken to achieve design goal **A** of Progger 3 are listed shortly in subsection 4.4.2.

To see why the measures given so far are necessary, consider a clean installation of Debian 10 GNU/Linux running without `lockdown` set. Any root user is able to insert arbitrary kernel modules, one of which may probe for Progger 3's secret key, or try to stop the two tasks Progger 3 runs (tasks are essentially processes). A root user could also use `kexec` to load a new kernel with an altered or absent Progger 3. Furthermore, if the `/dev/kmem` interface is available, one might have a chance of recovering Progger 3's secret key by reading from that interface, These attacks are all negated by booting with

`lockdown=confidentiality` [10].

However, using a trusted kernel can be a hindrance to the operation of some systems. For example, a user may be an administrator of their own personal computing device and want to be able to easily modify the kernel. For this user, the traditional trust boundary between regular users and root users may provide sufficient security. Progger 3 can still be used in this case, without any modification, so it not a requirement that users implement a trusted kernel if they decide it is unnecessary for their threat model. Of course, the tamper-resistance is lower in that case, as a compromised user space could, potentially, read kernel memory to find the encryption/authentication key, and then produce false records.

In contrast, some example deployments where these trusted-kernel requirements may be easier to satisfy are virtual machine deployments, and organisations issuing many devices to its members through an IT department. So, while the use of Progger 3's kernel-only mode with a trusted kernel is not feasible in every system, there are certainly areas where it can be used, and these areas are quite significant ones.

4.4.2 Kernel-only implementation in Progger 3

The Progger 3 client has no user space components. It can be compiled as a standalone kernel module, or a be compiled built-in to the kernel. The standalone module is intended for development and debugging, as it must be loaded by user space. Being loaded by user space, there will be a duration before the module is loaded where provenance is not collected, which is sufficient to disqualify the module from being able to collect provenance of all activity on a system. Being built-in to the kernel, which is a new feature of version 3 of Progger, means that every system call made can be logged, achieving design goal **G**. Furthermore, the built-in approach means that there is no risk that user space might be able to remove Progger 3 at run time. (Standalone modules can try to prevent user space from unloading them, but it is nice not

to have this risk at all by being built-in.) Hence, design goal **H** is achieved.

In addition to having no user space components, we spent time trying to ensure that there were no bugs in Progger 3's kernel code that could let user space programs influence Progger 3 in some undue way. An example of such an issue in Progger 2 is described in subsection 3.3.1.

Unfortunately, there are two issues related to user space being able to maliciously affect the availability of the data generated by Progger 3. These are listed in subsection 7.3.2 and subsection 7.3.3 These issues stem from the fact that a root user has some influence over task priority and network interfaces. To the best of our knowledge, fixing these issues would require quite invasive changes to Linux, and is far outside the scope of Progger 3.

Still, given the measures in place, user space is unable to modify the code of the Progger 3 client or any data produced by the Progger 3 client. As discussed earlier, in section 4.2, user space cannot change the configuration of Progger 3. So, design goal **A** is met.

That this kernel-only mode provides protection of Progger 3's data from user space is essential for Progger 3's use of a TPM, as will soon be described. The TPM is used to seal a cryptographic key, and when that key is loaded into Progger 3, the kernel-only mode means that user space cannot access the key.

4.4.3 Conclusion

Progger 3 has a true kernel-only mode, while Progger 1 and Progger 2 do not, as described in chapter 3. This means that only Progger 3, being built-in to the kernel, can ensure comprehensive logging of provenance, from the moment user space starts. It further means that Progger 3 cannot have its code or data altered by user space. Having a true kernel-only mode paves the way for Progger 3's enhanced tamper-resistance relative to its predecessors through the use of a TPM, which is detailed shortly.

4.5 Cryptography

Progger 3 is the first implementation of Progger to provide strong cryptography to ensure that the collected provenance is confidential and cannot be tampered with in-flight, and to allow the receiver to verify the provenance truly came from the expected provenance client.

In contrast, Progger 2 does not provide any assurances of confidentiality or integrity: data is sent in plaintext and unauthenticated.

With Progger 1, there is a proposed framework that provides confidentiality and integrity [3]. This framework is discussed in detail in subsection 7.2.1. What is relevant here is that this framework describes how to combine an existing provenance client with other software and a TPM in order to achieve confidentiality and integrity. Meanwhile, Progger 3 makes these improvements in the provenance client itself. When looking at Progger 1 by itself, as described in [2], the provenance records contain hashes based upon the last provenance record of the same system call. This is useful for detecting whether a malicious entry may have been added locally, but does not allow a receiver to authenticate that the provenance as a whole has not been tampered with in-flight. Consider, for example, an attacker in the network path that can intercept and modify every message sent by Progger 1. This attacker could modify all of the hashes so their alterations to the provenance would go undetected. Furthermore, there is no confidentiality. This may be because the envisioned usage of Progger 1 is within a cloud environment, where the network paths from systems running the Progger client to a server collecting the data are trusted. However, Progger 3's approach means that it can be used in many more scenarios, not just environments with trusted networks. It also means that tampering with the provenance is essentially impossible, assuming trust in the kernel, as opposed to being only "very difficult" in Progger 1 [2].

The following sections explain why confidentiality and integrity are essential. Then, the sections after that cover the measures implemented in Progger 3 to achieve confidentiality and integrity.

4.5.1 Confidentiality

Progger 3 can be configured to collect very detailed information about a system. This information can include the file names on a system, the time of each file access, as well as the programs being executed. So, by having access to the information generated by such a configuration of Progger 3, one could readily determine, in real-time or retroactively, what activities are, or might be taking place, on the system.

4.5.2 Integrity

An attacker in the network path between the Progger 3 client and server is able to modify any packet it forwards. Without any action to prevent it, this could give an attacker the ability to spoof packets, asserting that a phony action occurred; to modify packet contents, potentially hiding activity; or to drop packets entirely without the server knowing, again masking activity. Since there must be a high level of confidence in the collected provenance, such attacks must be thwarted.

4.5.3 Cryptography approach in Progger 3

In Progger 3, both confidentiality and integrity can be assured. In kernel mode, a complete TLS implementation is not available [2]. Such an implementation would be a serious undertaking, and likely add significant complexity and attack surface to the kernel [24]. Instead, Progger 3 uses a simpler approach based on XChaCha20-Poly1305.

When a message is to be sent over the network, it is encrypted and authenticated with XChaCha20-Poly1305. This process also binds some plaintext, known as the associated data, to the cipher text. When decryption takes place, the message and the associated data must both be untampered for ver-

[2]There is a feature of Linux called "kernel TLS", but that deals with data encryption only; the more complicated handshake is left to user space [23].

ification to succeed. The format of this message, as it appears “on the wire”, can be seen in Figure 4.5.

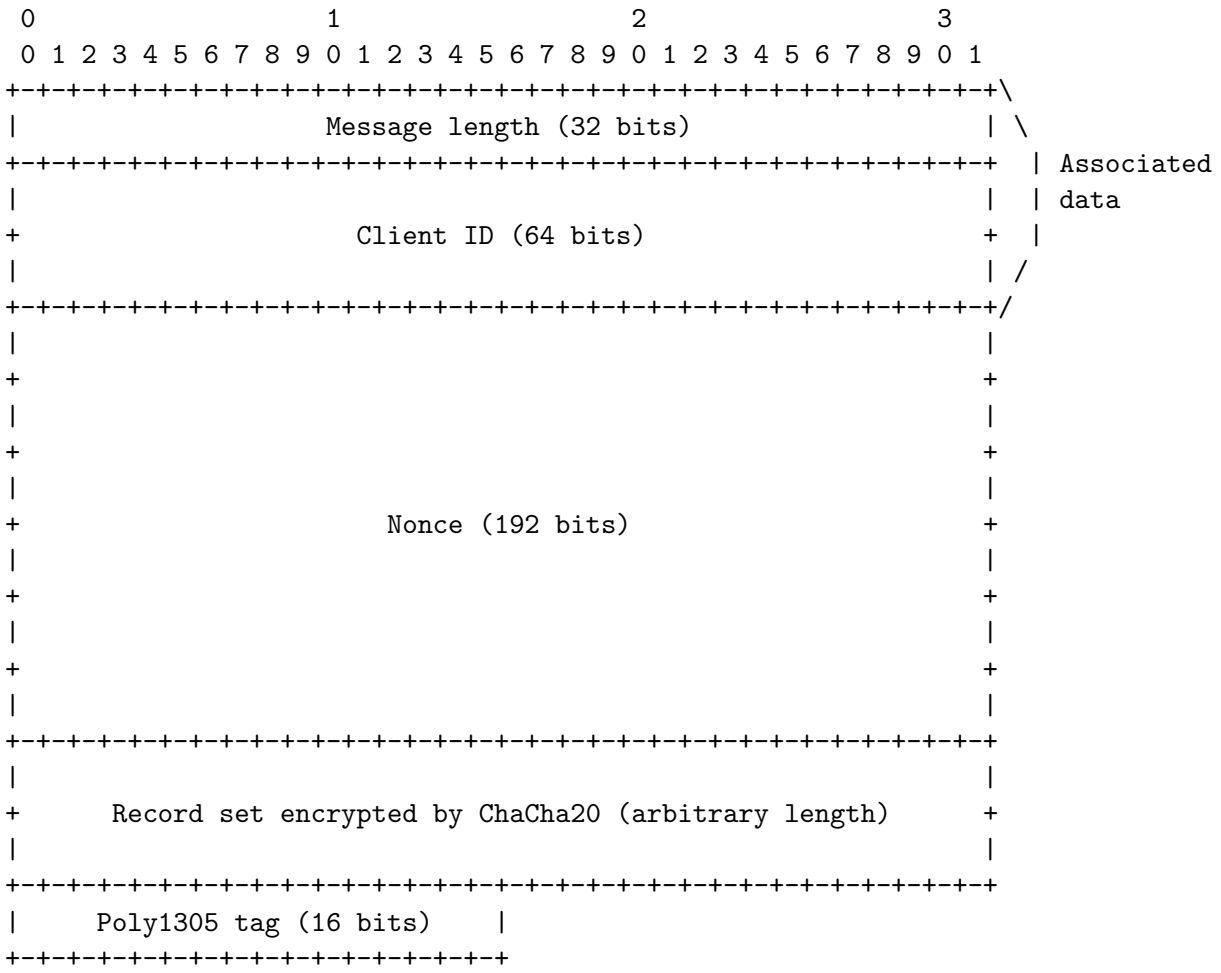

Figure 4.5: Format of messages sent over the network by Progger 3

The algorithm is performed by the function `xchacha20poly1305_encrypt`, which is part of the Linux kernel’s crypto library. Progger 3 does not have its own implementation of any cryptographic algorithm; instead, it uses existing implementations that have been reviewed by many people.

The reason that XChaCha20-Poly1305 was chosen over ChaCha20-Poly1305 is due to the longer nonce in XChaCha20-Poly1305, which offers a 192-bit nonce as opposed to a 96-bit nonce [12]. Given that the keys used on each system running the Progger 3 client are static, as is explained later, a simple counter nonce is unsuitable, as that would lead to reuse of {key, nonce} when the system reboots. Such reuse is to be avoided at all costs, as it reveals

the XOR of the plaintexts [11]. Use of a random nonce for each message is also unsuitable, due to the overhead of repeatedly generating a random nonce. Progger 3 opts to construct the nonce by concatenating a random value R, generated once when Progger 3 starts, with a counter n that increments once with each message. 96 bits is not enough to allow for a large enough R and n, but 192 bits is. Progger 3 uses R for the first 128 bits of the nonce, and n for the final 64 bits. This construction is not security-affecting compared to using an entirely random nonce, or an entirely counter-based nonce [12]. With 128 bits for R, it would take $2^{128/2} = 2^{64} \approx 1.84 \times 10^{19}$ reloads of Progger 3 for there to be a 50% chance of reuse of R. Further, Progger 3 would have to send 2^{64} messages without the system rebooting before the counter cycles, which, at an absurd rate of 1,000,000 messages per second would take over 500,000 years. So, nonce reuse will not occur with this construction.

The associated data used in Progger 3 is simple: a 32-bit unsigned integer indicating the length of the message, so the server receiving the message knows when it has a complete message, and a 64-bit unsigned integer to act as a unique client ID, as seen in Figure 4.5. Ideally, Progger 3 would not reveal the client ID. Yet, it seems to be the best approach, given that each client should have its own unique key used for encryption, and the server needs to be able to determine which key to use to decrypt each message it receives.

The message length placed in the associated data, unfortunately, needs to be used by the server before the message is authenticated, as the whole message must be received before it can be authenticated. If a malicious actor changed this value, it could cause Progger 3 to read too much or too little data. Upon receipt of the incorrect amount of data, the message would not authenticate, so it could be determined that the message was tampered with, but there would be no way of knowing what length of data is out of sync. So, Progger 3 would have to reinitiate the connection, losing some data in the process. However, this requires a network carrier to perform the attack, and they could simply drop part or all of the traffic anyway. So, effectively, there is no difference in

security due to the fact that Progger 3 uses the length value in the associated data before it can authenticate it. That is, as long as Progger 3 sanitises repeated extremely large values that might cause memory exhaustion. In the end, this in no way deals with a compromise of confidentiality or integrity.

It is also vital to avoid replay attacks and detect dropped messages. This could be accomplished by adding a message sequence number to the authenticated data. However, a message sequence number already exists: it's the counter in the nonce. So, if the server successfully decrypts a message, it knows that the nonce is correct, and it can use the embedded sequence number to check if the message has been replayed, or potentially determine if a message has been lost in transmission.

When considering confidentiality, one should consider how the message length might reveal information about its contents. For example, if an attacker knows that the messages being sent are records generated by Progger 3 for the `openat` system call, the message length could reveal the length of the path of the file opened, assuming the message contains only a single `openat` record, and no padding is added to the message. To avoid this information leak, Progger 3 pads its messages to a multiple of 16 bytes.

With all this considered, one can be assured that messages transferred by Progger 3 are done so with confidentiality and integrity. This means that design goals **C** and **D** have been met.

4.5.3.1 Private key storage

In order to utilise XChaCha20-Poly1305, Progger 3 requires a symmetric key that both the client and receiving server know, but which is kept secret from user space. If user space could access the key, it could forge messages. These forged messages could potentially arrive at the receiving server before the real messages, causing the real messages to be discarded in favour of the forged messages. This would mean that design goal **B** would not be met.

So, to prevent user space from being able to access the key, the approach

taken by Progger 3 is to use a key that is sealed by a TPM when the system is provisioned. A TPM policy is used so that the key can only be unsealed when a chosen PCR is in a specific state. When Progger 3 has unsealed the key, before user space has had an opportunity to execute, the chosen PCR is extended so that the key can never be unsealed again until the system's power is cycled.

The key would also be stored on the receiving server. However, it is assumed that user space is trusted on the receiving server, so the key can be stored without precautions such as the use of a TPM. As long as the disk the key is stored on is encrypted, and the key is only readable by the appropriate user space processes, the key on the receiving server can be considered secure.

This approach has the following disadvantages:

- A TPM is required.
- Extra work has to be done to seal the key when the system is provisioned.
- The key is also stored on the receiving server. If it is compromised from there, a client could be impersonated.
- A key compromise is more difficult to recover from, as the replacement key has to be re-sealed with the TPM.

Another approach that might, at first, seem a reasonable solution, is to use an ephemeral Diffie–Hellman key exchange. But it is essential that the receiving server can verify that it is communicating with the kernel client of a particular system, and the ephemeral Diffie–Hellman key exchange does not, on its own, achieve that. In order for the kernel client to authenticate itself to the receiving server, it needs a private key that must, again, be kept secret from user space. This ends up back at the original problem of requiring a method to keep a persistent secret hidden from user space.

However, this does raise a question: what if Progger 3 sealed the private key for client authentication with the TPM and used a Diffie–Hellman key

exchange to generate the XChaCha20-Poly1305 key? In this case, the private key couldn't be compromised from the receiving server, as it would only store the public part of the client key. This does offer a slight security benefit, but the following are disadvantages:

- The Progger 3 kernel client would have to process data that it receives from the network, increasing the kernel attack surface.
- There is no existing interface for directly performing a Diffie–Hellman key exchange in the Linux kernel. Having to implement this would be a considerable undertaking.
- The communication protocol between the Progger 3 kernel client and receiving server becomes more complex. Instead of the client simply sending a single type of message to the receiving server with no associated state (except a sequence number), the client and receiving server have to carefully keep track of what state they are in so that they can determine the correct type of message to send.

As the disadvantages were deemed to outweigh the advantages, it was not considered appropriate to utilise an ephemeral Diffie–Hellman key exchange in the kernel client of Progger 3.

A TPM is not the only way to store a key on a system such that it is only accessible to the kernel. This thesis focuses on the TPM approach, as that is what Progger 3 implements, but it is worth noting that the key could, for example, be stored on read-protected flash storage. As such, it is possible for Progger 3 to be modified to run on systems without a TPM and still provide its highest guarantees of tamper resistance.

4.6 Trusted platform module

As just described, Progger 3 makes use of a TPM to store the secret key used for XChaCha20-Poly1305. The following sections explore how Progger 3 utilises

a TPM and what requirements exist. A proof of correctness of Progger 3's TPM operations can be later found in chapter 5. The reader may be aware that the usage of a TPM with Progger 1 was previously explored by M. M. M. Bany Taha [3]. Compared to Progger 3, the reasons for TPM usage and the way in which the TPM is used are quite different. A comparison can be found in subsection 7.2.1.

4.6.1 TPM provisioning

When a system that is to use Progger 3 is provisioned, a symmetric key that is unique to the system must be generated and then sealed with the TPM of the system. Progger 3 requires that the TPM support version 2.0 of the TPM specification [5], and version 2.0 of the TPM specification is assumed for all the discussion here.

When a TPM seals data, the final product is an object, split into a public and encrypted private part, that can be used in combination with the TPM to recover the data. This object is known as a *sealed data object* [5]. The data itself isn't persistently stored in the TPM, nor are the parts of the sealed data object. To unseal the data, the sealed data object parts must be reloaded into the TPM, and then the TPM must be instructed to perform the unsealing.

TPM commands are very low-level, having many variables with very specific requirements. Few would construct these commands manually; instead, software can be used to abstract them away to more simple commands. For example, `tpm2-tools`[25]. There are some choices the provisioner must make when using `tpm2-tools`[25]. The requirements for these choices are listed shortly.

We have created a script, which is provided in the Progger 3 source code, that performs the provisioning using `tpm2-tools`[25], meeting all of the requirements listed below. It can be useful for further understanding of the steps required. It also ensures that the symmetric key never touches the disk of the system that is being provisioned. It may be found in section A.22.

To begin the requirements, the parent key used for sealing the symmetric key must:

- Be a persistent object
- Be a child of the storage root key (SRK), or be the SRK itself
- Specify strong encryption algorithms. Strong is subjective, but SHA256 and RSA2048 can be considered strong at this point in time.

Additionally, the lockdown and owner authorisation values must be set and kept secret. The last requirement is that a policy is added to the sealed data, so that it may only be unsealed when a chosen PCR is in a specific state, such as its default state.

The reasons for all of these requirements are made clear in the proof of correctness chapter.

Reprovisioning is possible by using the owner authorisation value, but care should be taken to ensure there is no malware on the system that might be able to intercept the authorisation or new symmetric key and use it later maliciously.

4.6.2 TPM unsealing

Once the TPM has been provisioned, the public and private part of the sealed data object, as well as the chosen PCR, can be provided to Progger 3 during compilation. Then, whenever Progger 3 starts, it performs algorithm 1.

Algorithm 1: `tpm_unseal_and_lock_key`

1. Load the sealed data object into the TPM.
2. Use the TPM to unseal the symmetric key.
3. Extend the PCR allocated to Progger 3.
4. Flush the loaded objects from the TPM.

If Progger 3 fails at any point, meaning that the PCR cannot be extended, which would leave the key potentially available to user space, a kernel panic

is induced. Any TPM object loaded by Progger 3 is also flushed in the error paths, so that the TPM does not reach its limit of loaded objects.

Having described the TPM operations within Progger 3, a proof its correctness can be found in chapter 5.

4.6.3 TPM benefits

Combining the use of a TPM with Progger 3's kernel-only mode means that a cryptographic key is available to Progger 3 and user space can never access it. This means that it is impossible for user space to create forged provenance records that would verify when decrypted by the remote server. Thus, the tamper-proof property extends to cover sending provenance over the network while in the presence of a malicious user space. This is the achievement of design goal **B**.

4.7 Performance improvements

The improvement to performance in Progger 3 is one of its significant advancements over Progger 1 and Progger 2. The improvements, in terms of benchmarks, can be found in chapter 6. This section focuses on the steps taken to achieve the performance increase.

The primary reason that a provenance system that monitors system calls would reduce system performance is that extra code is run each time a system call executes. This code does not run in parallel with the system call; instead, it runs as a step in the system call's execution, increasing the total execution time of the system call. With system calls being a common operation, this can significantly slow down many workloads if work is not done to minimise the length of time this extra code takes to run. Minimising this extra code run upon each system call is primarily the reason that Progger 3 is so much faster than its predecessors. Essentially, all Progger 3 does when a system call executes is collect information about the system call and write it to a ring

buffer. Any further processing, such as encryption and transferring the data over TCP, is done by a separate kernel task (which is essentially a separate process or thread). These separate kernel tasks run in parallel with other tasks on the system, so they do not directly add to system call execution time. As long as one ensures that these tasks do not use excessive CPU time, the overall decrease in system performance is not too severe.

As previously mentioned, one of the steps Progger 3 takes that adds to a system call's execution time is adding data to a ring buffer. As such, time has been spent ensuring that the ring buffer implementation in Progger 3 is efficient. One ring buffer is created for each CPU, to reduce contention, which naturally improves efficiency. Having to support elements of variable length means that the ring buffer is not entirely lockless, but care has been taken to ensure that the locks are held only very briefly. As a result, adding data to the ring buffer is a relatively quick operation.

So, Progger 3 has achieved its efficiency by separating system call data collection from data processing and having code that works in an efficient manner. We consider Progger 3 to be the first iteration of Progger that can be reasonably used in a wide range of workloads without reducing system performance to an unacceptable level. So, design goal **E** is achieved.

4.8 Ability to trace any system call

Progger 3 is able to trace any Linux system call, of which there are over 300 on x86-64 [4] (the exact number depends on the kernel configuration). In contrast, Progger 1 is only able to trace 32 system calls [2], and Progger 2 only supports 23. This is because Progger 3 uses a single, simple function for handling every system call, while Progger 1 and Progger 2 both use a separate function for each system call that they support (although sometimes one function is used for multiple similar system calls).

The design of Progger 1, where system calls were wrapped by replacing the

address of the system call functions, naturally led to writing a new function for each supported system call. Meanwhile, with tracepoints, which Progger 2 and Progger 3 use, one callback function is executed upon each system call entry and another upon each system call exit. Details about the executing system call can be collected from inside the callback. Progger 3 provides a single function as the callback to the both the system call entry and exit tracepoints, and as such can trace any system call with a single function. However, the callback function used in Progger 2 for the system call entry tracepoint still ends up calling individual functions for different system calls.

So, how is it that Progger 3 uses a single, simple function to trace every possible system call? A lot of the information can be collected in the same way across different system calls. For example, the system call number, the system call return value, process and user IDs of the currently executing task. But there is some variation in, for example, the system call arguments needing to be collected. Progger 3 tries to keep the management of these differences simple by using a single code path for each system call. It does this by keeping a table with a small amount of metadata about each system call. Most of this metadata is concerned with which of the arguments are C-strings and, in fact, most system calls do not need any metadata added to this table. This metadata doesn't need to include the number of arguments to the system call. Copying the maximum of six arguments each time ensures that each system call will always have all its arguments copied, and is probably even faster than trying to copy exactly only the number required, as that increases the size of the metadata table that has to be loaded into cache. Any excess arguments can be ignored when the data is processed later by other programs. With this approach, a single code path can deal with tracing any system call, as can be seen in section A.20, particularly the `syscall_tp` function.

It is worth pointing out that Progger 3's `syscall_tp` function, which deals with tracing each system call, is not very long or complex. In fact, it is only 45 lines, excluding whitespace. So, in addition to the extra usability of being

able to trace any system call, the single code path massively reduces code complexity and increases maintainability.

Naturally, design goal **F** is met.

4.9 Stability

During the development of Progger 3, extensive care was taken to ensure that it can be run for extended periods of time without crashing or harming the system. Having completed development, no system instability has been observed with the completed version of Progger 3 loaded. Further, no warnings or errors from any kernel subsystem were printed to the kernel log [3] (as read with `dmesg`), even when running on a kernel with many of the debug options enabled under the "kernel hacking" configuration section.

We estimate that Progger 3 has been run for at least two hundred hours during testing, with the longest single run perhaps being eight hours. Progger 3 is likely to be able to run for much longer than eight hours, however; perhaps months or years. In addition to the runtime testing, we carefully checked the code itself for correctness.

This improvement is quite noticeable when comparing with Progger 2. In the tests presented in chapter 6, Progger 2 caused a lot of system instability. This instability prevented us from being able to measure Progger 2's performance to the same accuracy as other programs.

Hence, design goal **I** is met.

[3] The only exceptions to this were that Progger 3 warned when running without using the TPM—which is benign when done intentionally during testing—and that some ring buffer overflows occurred, as sometimes the test server didn't receive the data sent by Progger 3 fast enough, due to data reception and processing being on the same thread in the server.

4.10 Maintainability

While not explicitly listed in the design goals of Progger 3, as it is a somewhat-difficult metric to quantify, part of Progger 3's architecture is to have a maintainable code base.

Progger 3 was developed against Linux 5.8.y. It very slightly modifies two files already present in Linux 5.8.y: `Kconfig`, and `drivers/net/Makefile`, adding just three lines of code to each. Then, it rewrites the `README` file to give information on Progger 3. The rest of the added code is self-contained. So, rebasing against later versions of Linux should be straightforward.

The patch in this thesis implementing Progger 3 adds 2465 source lines of code, 130 lines of comments, and 655 blank lines. This includes the kernel client, the server, and the TPM provisioning scripts. As such, with a small amount of code, maintenance should be relatively easy.

Additionally, since Progger 3 uses a single code path to process each system call, it is straightforward to make changes to the data that is gathered for each system call. Subsection 4.3.2 explains why the timestamp collected by Progger 3 uses nanoseconds since system boot. If a user decides that the system clock's value would better suit their use case, the change could be made simply, only needing to change one line in a single location, as the diff in Listing 4.2 shows.

Listing 4.2: Changing Progger 3's timestamp collection

```
1  --- a/drivers/net/progger/kernel/tracepoints.c
2  +++ b/drivers/net/progger/kernel/tracepoints.c
3  @@ -251,7 +251,7 @@ static void syscall_tp(struct pt_regs *regs, u8
       tp_src)
4
5          data->nr = id;
6          data->tp_src = tp_src;
7  -       data->ts = ktime_get_mono_fast_ns();
8  +       data->ts = ktime_get_real_fast_ns();
9          data->ret = tp_src == TP_SRC_SYS_EXIT ?
               regs_return_value(regs) : 0;
10
11         data->pid = current->pid;
```

4.11 Conclusion

Progger 3 has an architecture with many parts, but still retains simplicity, with each architectural decision having been made carefully and purposefully. The final architecture achieves all the design goals laid out in section 1.1, with each individual design goal's achievement being described in this chapter. But, by meeting these goals, what exactly does Progger 3 achieve?

By ensuring kernel-only operation, along with the use of strong cryptography and a TPM, Progger 3 can provide tamper-proof logging of provenance. This prevents tampering of provenance, both on the system running the Progger 3 client, and while the provenance is in transit to another system. Additionally, the provenance is confidential while in transit. Although we could not guarantee high levels of availability in the Progger 3 client, due to system design outside the scope of Progger 3, one can still be confident that all collected provenance is correct.

Also, the message format that Progger 3 uses to send provenance to a remote server allows for future expansion. If desired, additional sources of provenance could be implemented, or existing sources could have more detail collected from them. All this can be done while allowing the servers receiving the provenance to maintain compatibility with older versions of the Progger 3 client. The message format is also designed so that data is transferred in binary form, reducing the bandwidth needed.

Furthermore, an efficiency-focused design means that Progger 3 can realistically be used under many workloads, as is explored in chapter 6. The focus on simplicity resulted in increased usability: one can trace any system call; expect continuous, error-free operation; and easily modify Progger 3 to suit their needs.

Chapter 5

TPM Usage Proof of Correctness

This chapter proves that Progger 3's use of a TPM prevents user space from retrieving Progger 3's symmetric key from the sealed data object. The proof makes the following assumptions:

- The lockdown and owner authorisation values are not known to an attacker, are infeasible to guess, and no traces of them were left on the system during provisioning.
- Progger 3 is compiled built-in to Linux, not as a kernel module.
- The TPM provisioning met the requirements in subsection 4.6.1.
- The TPM implements the TPM 2.0 library specification [5] and *PC Client TPM* specification [26] correctly.
- Linux's TPM 2.0 driver follows the specification correctly and doesn't attempt to restore state when no state has been saved.

For this proof, a power cycle is defined as the system changing from ACPI state S4 or S5 to state S0, so that it involves Linux going through a full reboot.

Of the values specified during TPM provisioning for Progger 3, let n be the index of the PCR, and h be the value of this PCR that is embedded in the sealed data object's policy. Note that Progger 3 enforces $n \in [8, 15]$.

Lemma 1: PCR[n] must equal h for the symmetric key to be unsealed.

As the symmetric key is sealed during provisioning, a policy is specified that states that PCR[n] must be equal to h in order for the TPM to unseal the object. As user space cannot obtain the owner authorisation value, it is unable to create an alternative policy that is valid in the context of unsealing the symmetric key. The result follows.

Lemma 2: Once PCR[n] has its value extended from h, it can only regain a value of h after the system power cycles.

According to part 1 section 11.6.2 of the TPM 2.0 specification [5], "[t]he two ways to modify a PCR are to reset it or Extend it".

First, consider extending PCR[n]. Let v be the current value of PCR[n]. As chosen during Progger 3's provisioning process, the hash algorithm associated with this PCR is SHA256. For PCR[n] to be extended to h, it follows from the definition of the extend operation [5] that one would have to find a string of data s such that SHA256($v \parallel s$) = h. Finding such an s would be finding a SHA256 preimage, which, with the current best cryptanalysis, would take $2^{254.9}$ operations [27]. This is infeasible, so, extending PCR[n] is not an avenue to obtain a value of h.

Now, consider resetting the PCR. If the reset value of PCR[n] is not h, then there is no issue. However, it is expected (although not required) that h will be the reset value, so it is important to consider how PCR[n] may be reset. Two of the ways in which a PCR may be reset are the `TPM2_PCR_Reset` command, and a D-RTM event [5]. However, Progger 3 only allows $n \in [8, 15]$. For each such n, PCR[n] cannot be reset by `TPM2_PCR_Reset` or a D-RTM event [26]. It should be noted that the guarantee that these PCRs are not resettable comes from the TPM PC client specification (table 6) [26], not the TPM library specification, so it is not necessarily true for all TPMs. But, by assumption, only a PC client TPM needs to be considered. The only other methods listed in the TPM library specification (part 1 section 17.1) [5] to reset a PCR is through a TPM Reset/Restart/Resume, which, as per the

TPM library specification (part 1 section 12.2.3.2) [5], respectively correspond to a system reboot, resume from hibernation, and resume from suspend. A system reboot and resume from hibernation require a power cycle, but a system resume from suspend does not. So, further investigation of the TPM Resume sequence is required.

Section 12.2.3.2 of the TPM library specification [5] defines a TPM Resume as a Startup(STATE) that follows a Shutdown(STATE). This section explains that sequence as calling `TPM2_Startup` with `startupType == TPM_SU_STATE` after `TPM2_Shutdown` was called with `startupType == TPM_SU_STATE`. There is a copy-and-paste error, and `TPM2_Shutdown` should instead be called with `shutdownType == TPM_SU_STATE`, corroborated by the fact that section 9.4 of part 3 of the TPM library specification [5] lists only a `shutdownType` parameter, not a `startupType` parameter, for `TPM2_Shutdown`. Turning to section 9.3.1 of part 3 of the TPM library specification [5], it is shown that a TPM Resume sequence results in the values of specified PCRs being saved and then restored, but the exact PCRs are determined by the platform-specific specification. Turning to the relevant platform-specific specification, the TPM PC client specification [26], Table 6 shows that PCRs 0–15 are preserved. So, finally, it can be seen that the TPM Resume sequence will not reset PCR[n], as $n \in [8, 15]$.

One may wonder what happens if Startup(STATE) occurs without a previous Shutdown(STATE). As per the TPM library specification (part 1 section 12.2.3.2) [5], `TPM_RC_VALUE` is returned. The PCR values will be undefined and may be cleared. This would provide an avenue for PCR[n] regaining the value h without going through a power cycle. However, it also depends on Linux incorrectly issuing the Startup(STATE) and Shutdown(STATE) sequences, which, by assumption, will never happen.

Having looked at all the methods to modify a PCR, it follows that PCR[n] can only have its value reset through a power cycle.

Lemma 3: Once Progger 3 has initialised, PCR[n] will always have its

value extended from h.

If Progger 3 is unable to unseal the key and then extend PCR[n], it will induce a kernel panic. Thus, PCR[n] will always be extended by the time user space runs. The extension must be from h, as if the value of PCR[n] were not h, the unseal would fail so Progger 3 would cause a kernel panic.

Theorem: User space cannot retrieve Progger 3's symmetric key from the sealed data object.

When Progger 3 is in kernel-only mode, it will always be initialised before user space runs. From lemmas 1, 2, and 3, it follows that a power cycle is required for the object to be unsealed again. But a power cycle would result in Progger 3 reinitialising before user space has any chance to unseal the symmetric key. Thus, user space cannot unseal the symmetric key.

All that remains is to verify that the sealed data object stores the symmetric key securely at rest, and not in an unencrypted form. The public part of the sealed data object contains metadata about the object, as described in the TPM library specification (part 3 section 12.1, along with part 2 section 12.2) [5]. Furthermore, part 3 section 12.1 and part 2 section 12.3 of the TPM library specification [5] show that the private part consists of an integrity hash, along with an encrypted sensitive area. For the following TPM specification references, ensure that you are reading the version that contains the code samples. The code sample in part 3 section 12.1.3 [5] shows how the encryption is applied to the sensitive area. This code reveals that the private part, referred to as `outPrivate`, is created in the `SensitiveToPrivate` function. Using part 4 of the TPM specification [5], it follows that the sensitive data area is encrypted with the algorithm from the parent key supplied to `TPM2_Create`. As Progger 3's provisioning process requires the use of strong encryption algorithms for the parent key, it follows that the sealed data object does protect the symmetric key at rest.

Hence, user space cannot recover the symmetric key from the sealed data object.

Chapter 6

Evaluation

This chapter evaluates Progger 3 by determining what performance impacts may occur under a wide range of workloads. That is, it seeks to evaluate how much Progger 3 slows these workloads down. Rather than testing a multitude of individual workloads, this evaluation seeks to provide an understanding at a more general level about what impacts Progger 3 has on system throughput and latency. Then, by knowing where the bottlenecks are in specific workloads, one can have a good idea of how Progger 3 will impact performance under that workload. The main tests used for this evaluation are: compiling Linux to determine the impact on system throughput, and running a micro-benchmark of the `openat` system call to determine changes in latency. The list of system calls traced will also be varied to understand how the performance impact can vary when many or few traced system calls occur.

This evaluation compares Progger 3 to other programs. Some of these other programs may have scope greater than just a provenance system; so, the programs are collectively referred to as (system) tracers, of which provenance systems are a subset.

This evaluation compares tracers that are able to monitor the entire system call activity on a system and selectively choose subsets to trace, as this is exactly what Progger 3 does. Some of the provenance systems mentioned in chapter 3 do not have the ability to do this, so their performance is not

evaluated here. This evaluation only tests programs that can produce (nearly) the exact same output as Progger 3.

The following evaluations were performed on a bare-metal system running on an Intel i3-2120 CPU at 3.30GHz, microcode revision 0x2f. It had 7.73 GiB of DRAM available. The system was running Debian GNU/Linux 10 (Buster). The kernel running was Linux 5.8.18, with the patches adding Progger 3 applied.

In each test, the version used of Sysdig was `0.27.1`, the version of SystemTap was `4.3-119-gacd978b01` (as described by `git`), and the version of bpftrace was `0.11.2`.

6.1 Impact on system throughput

Having a tracer running on a system will occupy some of that system's resources. As such, there will be fewer resources for other programs to make use of. Fewer resources available means that programs are likely to experience reduced throughput; that is, programs will take longer to process a certain amount of data, or, worded differently, will be able to process less data in a given time period.

For a given program, its throughput is constrained at any specified moment by a single bottleneck. These bottlenecks could be the available CPU time, available memory, disk read/write speed, or available network bandwidth, to name some common ones.

CPU bottlenecks are quite common, and so are what this evaluation focuses on first. The disk read/write speed is not relevant to Progger 3. Progger 3 does not require a considerable amount of memory, so it is not meaningful to pursue an analysis of memory-bottlenecked workloads. Network bandwidth is relevant, as Progger 3 sends its output over the network, so this evaluation spends some time analysing Progger 3's impact in this regard.

6.1.1 Impact on CPU throughput

As each tracer operates, it takes CPU time to process each system call. If a system mostly has spare CPU capacity, the impact on throughput may be negligible. Yet, in situations with no spare CPU capacity, each moment a CPU spends executing a tracer is time that another program has lost.

If a tracer is opting to trace more system calls, or the system calls being traced occur more frequently, the tracer will spend more CPU time processing the system calls, reducing the throughput of CPU-bound programs. The frequency of system calls not being traced is of less importance: if a tracer is notified of a system call occurring that it isn't tracing, it should not spend any time processing it, so only a small amount of time would be spent in determining that the system call is not to be traced.

So, we conducted tests to assess the throughput of some workloads that make full utilisation of available CPU time. The results of these tests are presented in the following sections. The first test is compiling a Linux kernel while tracing some common system calls. The second test repeats compiling the Linux kernel, but traces only a system call that never occurs. The third test is a program that utilises all free CPU time without making system calls.

Given that the throughput impact depends only on the frequency of system calls and the portion of system calls traced, these tests should give a reliable overview of each tracer's impact on system throughput where the CPU is the bottleneck.

6.1.1.1 Time to compile Linux 5.8.3

In this test, Linux 5.8.3 was compiled 48 times per tracer evaluated. The `.config` file was generated by running `make allnoconfig`. Compiling Linux involves lots of system calls being made, such as `openat`, `read`, and `write`.

With each tracer tested, the following system calls were traced: `open`, `openat`, `openat2`, `rename`, `renameat`, `renameat2`. There were three reasons for this selection. First, the system calls should occur frequently during the

test, and `openat` certainly does. Second, they should be system calls that have C-strings as arguments, but not any other pointers as arguments. This is because Progger 3 copies C-string arguments, such as file paths, but no other pointer values. Copying C-strings is a relatively expensive operation, so it is important to account for its overhead in this test. Furthermore, this selection of arguments ensures that the comparison between tracers is fair, as it means that no tracer is copying arguments that Progger 3 is not. Third, only a small number of system calls are traced, to reflect what may be a reasonable real-world use of tracing. It is unlikely that one would want to collect information about every read and write performed on the system, but recording all files openings is somewhat more realistic.

Continuing with pursuing fairness, the output of each tracer has to be directed somewhere. Progger 3 encrypts its output and sends it over TCP, and is the only tracer of those tested in this evaluation with a built-in method of doing so. Instead of using separate programs to encrypt and transfer the output over TCP, each tracer other than Progger 3 simply had its output directed to `/dev/null`. This is so the results are not influenced by the efficiency of external programs used for encryption and transferring the output over TCP; rather, the results reflect only the actions performed by the tracer itself. This does give Progger 3 a slight disadvantage, but, as the results will show, Progger 3 still manages to be comfortably faster than the rest. There is little need for this analysis to determine exactly how much greater that margin could be.

Each tracer has its own mode of operation, so a list now follows providing any notable information about the use of each tracer in these tests.

Progger 3 — The output was sent via TCP to a physically separate system, so the system running the Progger 3 client did not incur any undue overhead from running the Progger 3 server. It was also confirmed by checking the output of `dmesg` that Progger 3's ring buffer never overflowed.

Sysdig — The following command was used to run Sysdig:

```
1 sudo ./userspace/sysdig/sysdig -p '%evt.arg.name' evt.type=open or
```

```
   evt.type=rename or evt.type=openat or evt.type=renameat or
   evt.type=renameat2 or evt.type=openat2 >/dev/null
```

SystemTap — The following command was used to run SystemTap:

```
sudo stap syscall.stp >/dev/null
```

where `syscall.stp` is

```
probe
       syscall.open,
       syscall.openat,
       syscall.rename,
       syscall.renameat,
       syscall.renameat2
{
       printf("%s\n", argstr);
}
```

The system call `openat2` was not supported by SystemTap at the time of testing. However, `openat2` was never called while compiling Linux.

bpftrace — The following command was used to run `bpftrace`:

```
sudo bpftrace bpftrace-probes >/dev/null
```

where `bpftrace-probes` is

```
tracepoint:syscalls:sys_enter_open {
       printf("%s\n", str(args->filename));
}

tracepoint:syscalls:sys_enter_openat {
       printf("%s\n", str(args->filename));
}

tracepoint:syscalls:sys_enter_openat2 {
       printf("%s\n", str(args->filename));
}

tracepoint:syscalls:sys_enter_rename {
       printf("%s %s\n", str(args->oldname), str(args->newname));
}

tracepoint:syscalls:sys_enter_renameat {
       printf("%s %s\n", str(args->oldname), str(args->newname));
}

tracepoint:syscalls:sys_enter_renameat2 {
       printf("%s %s\n", str(args->oldname), str(args->newname));
}
```

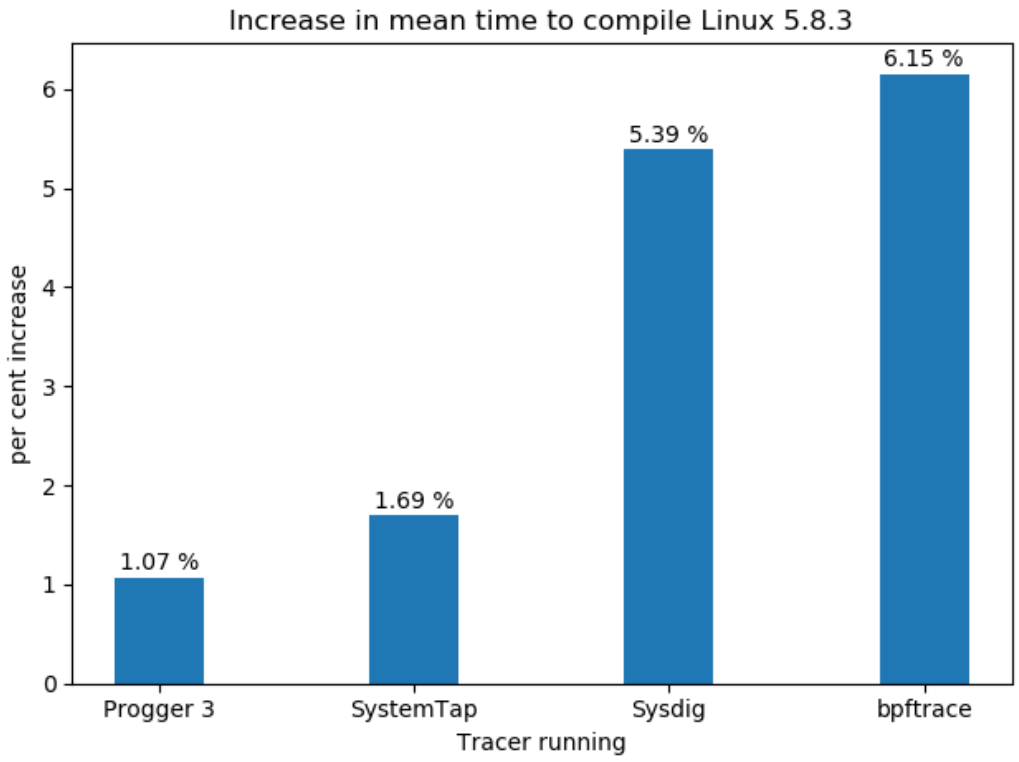

Figure 6.1: Linux compile test results

Each sample in this test is the real time for a single Linux compile. Before collecting the samples, Linux was compiled once with `make -j6 bzImage`, so that further compilations would be using cached files, leading to more consistent results. The following command was used to collect the samples:

```
1 for n in {1..48}; do (make clean && time make -j6 bzImage); done
```

The results are presented in Figure 6.1 and Table 6.1. These results appear reliable, as shown by the small standard error in Table 6.1. Progger 3 performs excellently in this test, causing by far the smallest performance reduction.

Progger 1 was not tested, as, being developed against Linux 2.6.32, it no longer compiles on Linux 5.8.18 without significant changes. However, the latency impact of Progger 1 is evaluated in subsection 6.2.1.

There was one other tracer tested: Progger 2. Unfortunately, there were several issues that prevented an analysis as thorough as the other traces.

It should be noted that the source code of Progger 2 had to be modified

Tracer	Sample size	Sample mean / seconds	Standard error
None	48	61.427	0.010
Progger 3	48	62.084	0.010
SystemTap	48	62.468	0.015
Sysdig	48	64.736	0.019
bpftrace	48	65.205	0.014

Table 6.1: Real time to compile Linux

slightly to get it to compile with Linux 5.8.18, but this amounted to a Makefile change and switching to 64-bit time structures. Neither of these should meaningfully impact performance.

To start with, Progger 2 supports only the following system calls: `open`, `read`, `pread64`, `write`, `pwrite64`, `close`, `mkdir`, `rmdir`, `rename`, `dup`, `dup2`, `sendfile`, `link`, `unlink`, `unlinkat`, `symlink`, `chmod`, `fchmod`, `chown`, `fchown`, `lchown`, `pipe`, `pipe2`. This is not compatible with the system call set used for testing the other tracers, as `openat` is missing. Furthermore, `openat` is the only system call that was used for opening files when compiling Linux during these tests. This is because `open` has been deprecated and `openat2` has not seen widespread adoption yet. So, for this comparison, the system call set used is simply all of the system calls traced by Progger 2.

The second issue is that Progger 2 caused system instability. The system was prone to crashing, so only one sample was collected. Furthermore, `gcc` would often segfault while compiling Linux. The one sample collected is shown in Listing 6.1. As with Progger 3, the output of Progger 2 was sent over TCP to a physically separate system.

Listing 6.1: Linux compile time while running Progger 2

```
1 real    0m57.405s
2 user    3m0.655s
3 sys     0m33.194s
```

This is an interesting result, as it appears that Linux has been compiled

in a time *faster* than with no tracer running. The reality is that, since `gcc` terminated with a segfault many times, Linux was not properly compiled, and the lower time reflects that fact that less work was done. While the difference in real time is unsuitable for comparison, it can be seen that the system CPU time taken was 33.194 seconds, an increase compared to the mean of 21.951 seconds with no tracer loaded, and the mean of 22.149 seconds ($n = 8$) with Progger 3 loaded and tracing this new system call set. From this increase in system CPU time taken, we coarsely estimate that Progger 2 causes a 50% reduction in CPU throughput when compiling Linux.

It should be noted that Progger 2 sends somewhat different data to Progger 3. For example, Progger 2 always sends the inode number of files read and written. Still, there is an exceptional difference in performance between the two, to the point that one could reasonably expect that Progger 3 could be modified to collect the same data as Progger 2 while maintaining its performance lead.

6.1.1.2 Time to compile Linux 5.8.3 with no traced system calls occurring

In this test, Linux 5.8.3 was compiled in the same manner as in subsubsection 6.1.1.1. But this time, Progger 3 traces only `tuxcall`, which is an unimplemented system call that no longer has any legitimate use. This means that Progger 3 does attach to the `sys_exit` tracepoint, but its tracepoint handler exits very quickly, as Progger 3 never encounters a system call that it has been instructed to trace. Sysdig also traces `tuxcall` for this test. In contrast, bpftrace and SystemTap trace `sched_rr_get_interval`, as they do not support `tuxcall`. This difference isn't significant, as `sched_rr_get_interval` is never called during the test.

The results in Figure 6.2 and Table 6.2 show that Progger 3 is performing respectably in this metric, being on par with bpftrace and SystemTap, and indeed performing better than Sysdig.

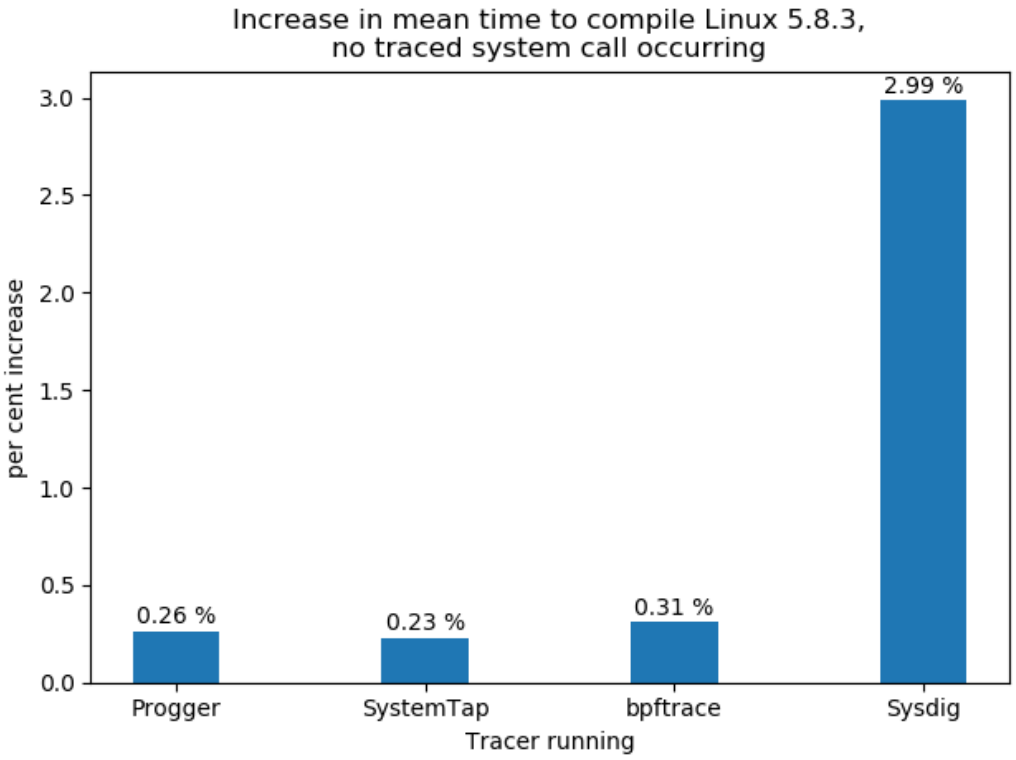

Figure 6.2: Linux compile test results

Tracer	Sample size	Sample mean / seconds	Standard error
None	48	61.233	0.009
Progger 3	48	61.395	0.013
SystemTap	48	61.374	0.011
bpftrace	48	61.423	0.014
Sysdig	48	63.062	0.010

Table 6.2: Real time to compile Linux, no traced system call occurring

6.1.1.3 Impact on a CPU-bound program making no system calls

In this test, the program shown in Listing 6.2 was run, with one instance for each available CPU:

Listing 6.2: Test program making no system calls

```
1 #include <stdint.h>
2
3 int main(void)
4 {
5   for (volatile uint64_t i = 0; i < 64ULL * 1000 * 1000 * 1000; i++)
6       ;
7
8   return 0;
9 }
```

This caused 100% CPU utilisation. Meanwhile, Progger 3 was loaded, and tracing the same system calls as in subsubsection 6.1.1.1.

The execution time was measured 48 times with the following command:

```
1 for n in {1..48}; do time ( for m in $(seq "$(nproc)"); do
     /tmp/no-syscall & done; wait ); done
```

After excluding extreme outliers that occurred both with no tracer running and with Progger 3 running, there was only a 0.01% difference in the mean execution time when Progger 3 was loaded, with a standard error of 0.049%. We conclude that there is no meaningful difference present in these results. Hence, there is no need to test any of the other tracers for comparison, as Progger 3 is performing ideally in this metric, so no other tracer could perform better.

6.1.2 Impact on network throughput

In this test, Linux 5.8.3 was compiled as described in subsubsection 6.1.1.1. But, this time, the test measures the network traffic generated. Performing the compilation once, 171 MiB of traffic was generated, which is an average throughput of 21.3 Mbps. This is quite a significant impact, and would not be desirable in many cases where the network is the bottleneck of a workload. Progger 3 tries to efficiently pack the data it sends, keeping data in binary form

instead of translating it to human-readable strings. So, to reduce the amount of data sent, there are two options: use compression, or further restrict what Progger 3 traces. Compression would involve extra CPU time, and, in many cases, this could nullify any benefit from the reduction in data sent. Still, it may be worth at least investigating in some cases. With regards to further restricting what Progger 3 traces, restrictions are currently limited to which system calls are selected to be traced. In the future, one could implement in Progger 3 the ability to restrict tracing of system calls based on the system call's arguments. For example, it could be enforced that `openat` would only be traced for paths under `/example/secret/dir`, which has great potential to reduce network overhead.

6.2 Impact on system latency

Progger 3 operates by attaching to Linux tracepoints, resulting in extra code being run when a system call is made. One would expect this to introduce additional latency to system calls. This section seeks to determine precisely how significant the latency increases of Progger 3 and similar system tracers are.

The latency increase may not be constant between different system calls. Some system calls may take less time to process, such as those with only integer arguments, having no C-strings to be copied. Furthermore, system calls may incur increased latency even when they are not specifically being traced. This can arise, as is the case with Progger 3, by the fact there are only two system call tracepoints available: one for system call entry, and one for the system call exit. That is, a tracepoint handler will always be executed when a system call runs, no matter the system call. If the system call is not being traced, then the tracepoint handler can be exited quickly, but some time will still be spent.

Given these facts, we conducted two tests. The first analyses the time

taken for `openat` to complete while it is being traced. This system call has a string argument specifying the file to open, so the latency increase will include the impacts of string copies. The file opened each time is `/dev/null`. The second test is similar, except it analyses the time taken for `openat` to complete while it is *not* being traced. For Progger 3 at least, one can expect the latency increase shown to be true of every system call not being traced, as Progger 3 uses a single code path for each system call.

The commands used to run each tracer were the same as given in the CPU throughput tests. It was also verified that Progger 3 did not experience any ring buffer overflows.

To reduce variance in the results, the following measures were taken: TurboBoost was disabled, giving a constant maximum CPU frequency; the program executing `openat` was pinned to a single CPU, which the program had exclusive use of (at least, among user space tasks, as kernel tasks cannot be arbitrarily prevented from running on specific CPUs), and any of that CPU's siblings were also reserved; the `performance` CPU governor was used, so that the CPU would always run at its maximum frequency; a high-resolution timer was used: a constant-rate time stamp counter (TSC); and `openat` was called 1,000,000 times before the timing started so that it would be warm/cached.

Time was measured in CPU clock cycles. This can be approximately converted to µs by dividing by 3,300, since the CPU these tests were performed on ran at a (near-)constant 3.30 GHz.

Unfortunately, Progger 2 could not be tested, as it would cause the system to crash as soon as the test started.

6.2.1 Impact on a system call being traced

This analysis begins by looking at a sample of 50,000 `openat` calls taken with no tracer running. To allow a graph of this sample to be readable, any value above the 99.9th percentile is excluded, amounting to excluding 50 of the samples. The five largest samples excluded are 1,091,496, 1,196,196, 1,256.292,

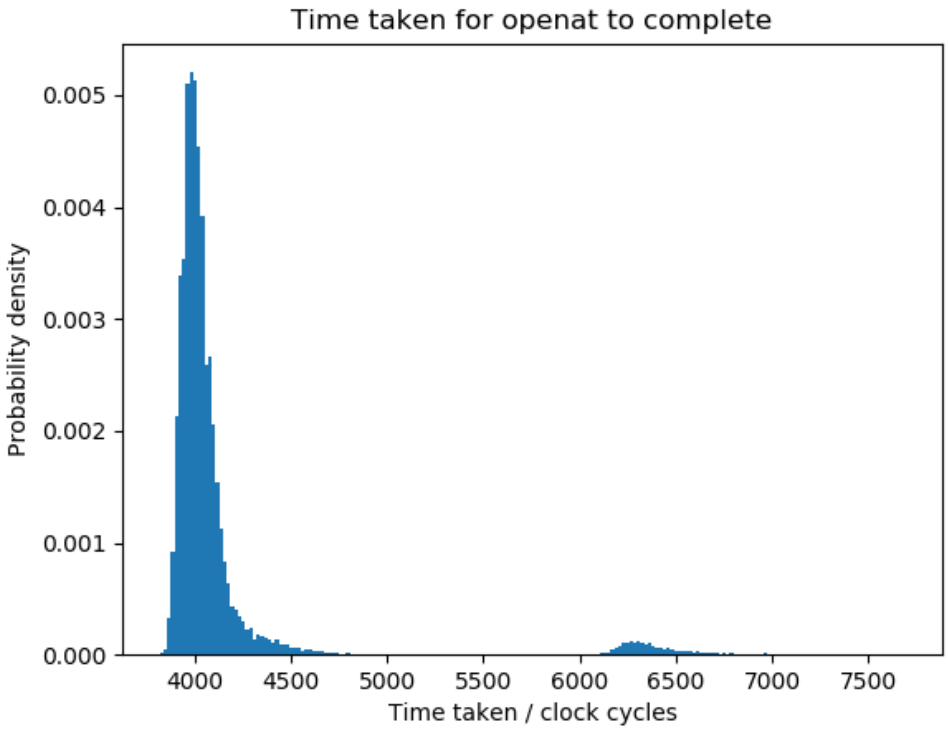

Figure 6.3: Time taken for `openat` to complete

1,795,448, and 1,797,332 clock cycles. This sample is presented in Figure 6.3. The distribution is bimodal and appears to be composed of two separate distributions, each non-normal with a right skew. Given this distribution, along with the extreme outliers excluded, it is clear that the mean is not an appropriate statistic for this analysis. Still, the median and 99th percentile of the samples can be calculated as reasonable indicators of performance.

For each tracer, 48 samples, $S_1, \ldots, S_{48}$, were collected, where each sample S_n is a set of 5,000,000 execution times of `openat`. This approach is taken so that, when the median and 99th percentile are estimated, the distribution of S_n gives some idea of what confidence can be placed in the results. To estimate the median and 99th percentile for each set of samples $S \coloneqq \{S_1, \ldots, S_{48}\}$, the following calculations are used:

$$\text{median}(S) = \text{median}(\text{median}(S_1), \ldots, \text{median}(S_{48})),$$

$$\text{percentile}_{99}(S) = \text{median}(\text{percentile}_{99}(S_1), \ldots, \text{percentile}_{99}(S_{48})).$$

In Figure 6.4, the error bars show the range of $\text{median}(S_n)$ for each n. Similarly, in Figure 6.5, the error bars show the range of $\text{percentile}_{99}(S_n)$ for each n.

Looking at Figure 6.4 and Figure 6.5, Progger 3 can be seen quite comfortably leading in terms of system call latency overhead. Furthermore, the error bars in Figure 6.4 and Figure 6.5, which represent the range of values, give confidence in these measurements, despite the unfavourable distribution of samples shown in Figure 6.3,

Finally, this metric can also compare the performance of Progger 3 to Progger 1. We could not test Progger 1 directly, as it would have to be modified too much to compile on Linux 5.8. However, the results from the paper outlining Progger 1 [2] lists execution time of calls to `open` increasing from 3.54 µs to 758.7 µs, a 21,000% increase, and calls to `write` increasing from 299.06 µs to 2072.08 µs, a 590% increase. Progger 3 performs well relative to this, with a median 10% increase in `openat` time when opening `/dev/null`. As the `read` system call involves no string copies in Progger 3, the increase

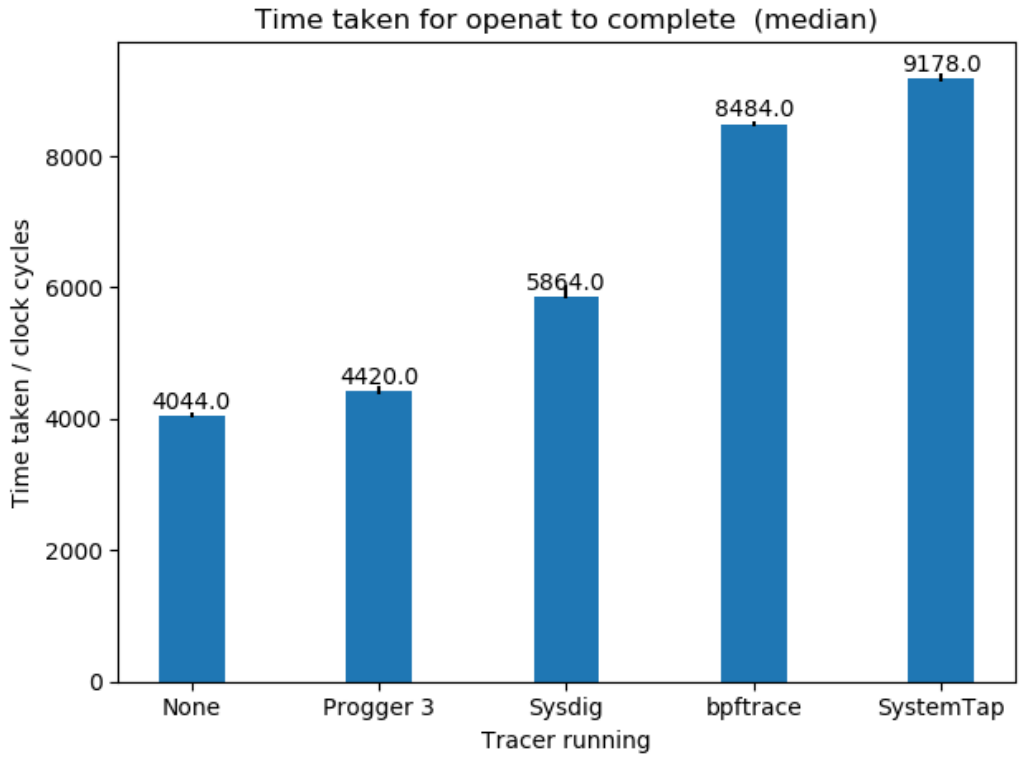

Figure 6.4: Median time taken for `openat` to complete

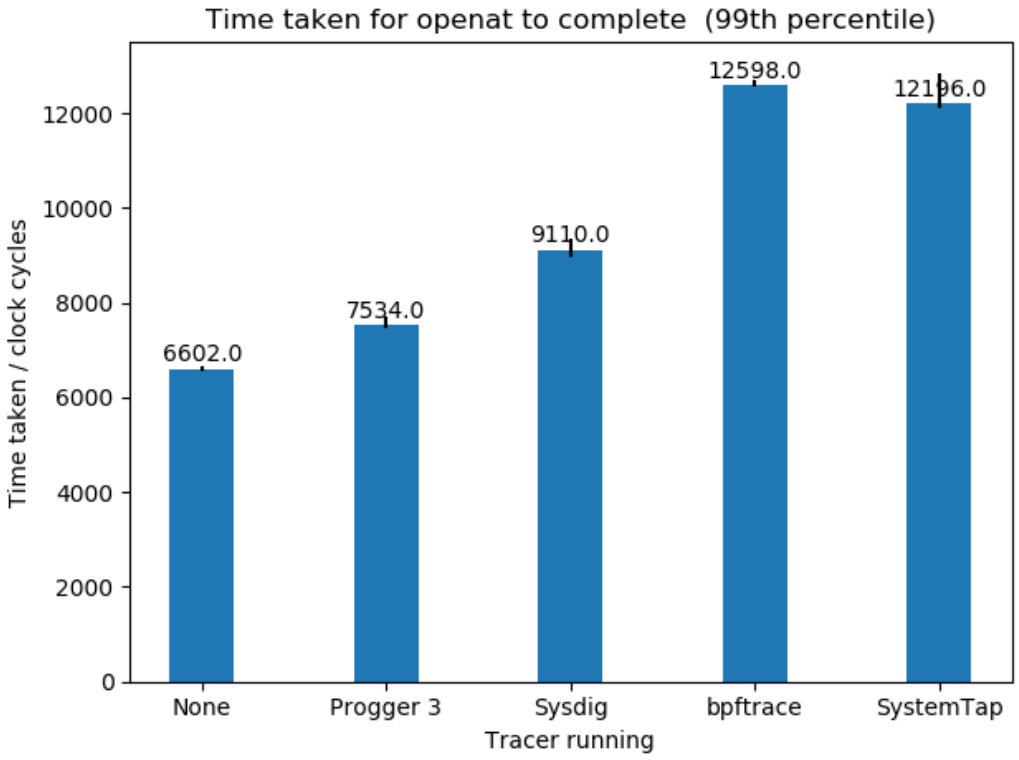

Figure 6.5: 99th percentile of the time taken for `openat` to complete

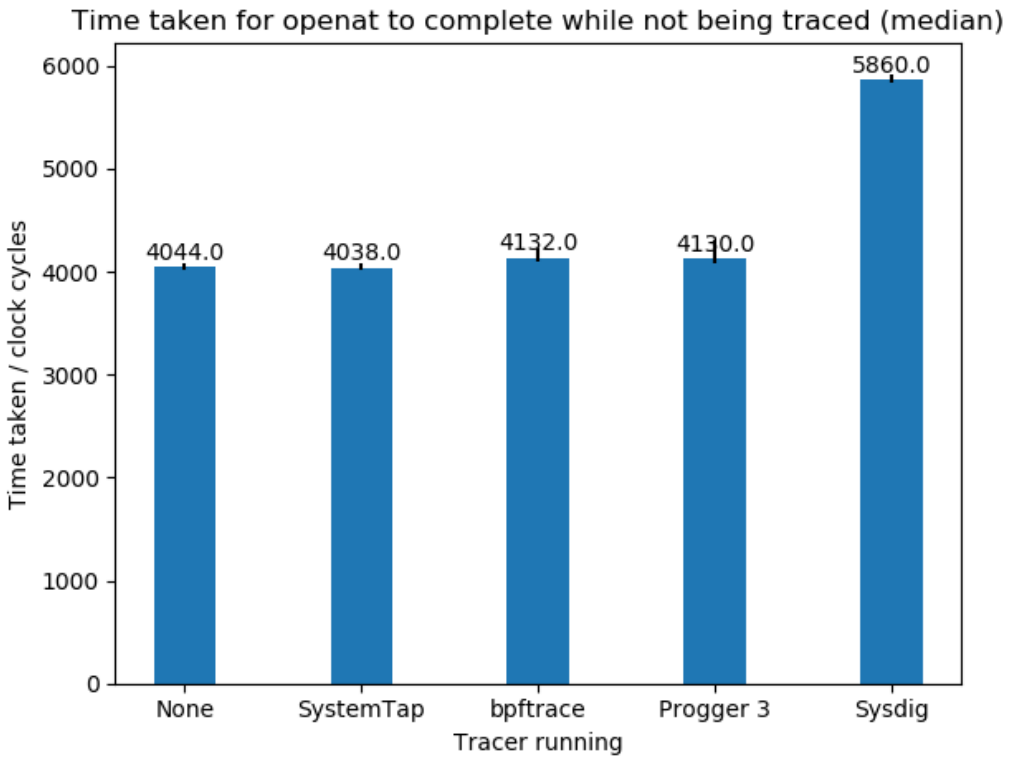

Figure 6.6: Median time taken for `openat` to complete while not being traced

in execution time would be even less than `openat` in absolute duration, and likely less in terms of percentage too.

6.2.2 Impact on a system call not being traced

For this test, as with subsubsection 6.1.1.2, Progger 3 and Sysdig trace only the system call `tuxcall`, and bpftrace and SystemTap trace only the system call `sched_rr_get_interval`. Again, it was verified that `sched_rr_get_interval` was never called during these tests.

Using the same methodology as in subsection 6.2.1, the latency overhead can be estimated, which is shown in Figure 6.6 and Figure 6.7. The results are similar to subsubsection 6.1.1.2. Progger 3 performs adequately, with approximately the same overhead as bpftrace. It is interesting to see that SystemTap shows no overhead. This is potentially because SystemTap uses kprobes to instrument the system calls, not tracepoints.

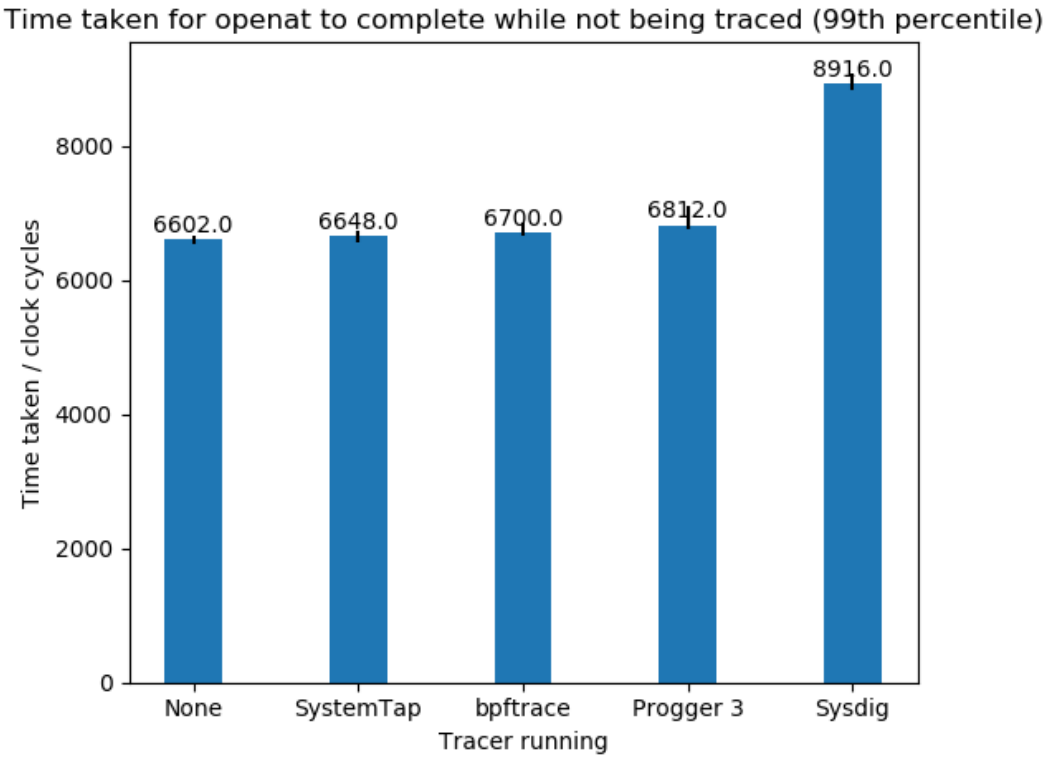

Figure 6.7: 99th percentile of the time taken for `openat` to complete while not being traced

6.3 Correctness

It would be for naught if Progger 3 were not producing correct results. To check for correctness, a program was run that opened each file found on the system. It then checked the output of Progger 3 to see if each file opening was logged, in the correct order, with the correct PID and return code. The program also opened a small random number of other files before opening the next file in its list, so that the file descriptor returned by `openat` would be varying considerably. This test program has been run several times, completing each time without error.

Furthermore, many less-structured observations were made of Progger 3's output, such as creating and renaming files, `ssh`ing in to the system running Progger 3, and background activity. Nothing incorrect was observed in these situations.

The TPM code paths, from provisioning to unsealing the secret key, have been tested with a virtual TPM 2.0 device, using `swtpm` [28] and `qemu` [29]. Again, these code paths worked without a problem.

We have no reason to suspect that there are any significant flaws in Progger 3's correctness.

6.4 Summary

These results are very favourable for Progger 3. They show that, while Progger 3 is loaded, the impact on system performance can be relatively small. So, Progger 3 could be reasonably deployed on systems that can afford slight drops in performance. For workloads that perform few to no system calls, the performance impact may be effectively zero.

Unfortunately, the network overhead can be quite high, even in somewhat reasonable configurations of Progger 3, such as tracing all file openings. This presents opportunities for future improvement: conditionally tracing system calls based on their arguments, and compression of transferred data.

While we found a precise comparison between Progger 1, Progger 2, and Progger 3 to be infeasible, the observations that we were able to make suggest that Progger 3 has achieved a significant increase in performance and system stability relative to its predecessors.

When it comes to Progger 3's performance relative to the other tracers tested, it should be noted that Progger 3 is developed with very different goals to the others. The other tracers are more like debugging tools, whereas Progger 3 is focused on being a provenance system. As such, it is perhaps not a surprise that Progger 3, designed to run continuously, as part of regular system use, is more performance-optimised.

Chapter 7

Discussion

7.1 Chapter outline

This chapter will discuss some miscellaneous topics relating to Progger 3, providing a better idea of where Progger 3 fits within the bigger picture.

Section 7.2 compares some aspects of Progger 3 to similar work, expanding on what sets Progger 3 apart. Then, section 7.3 explores some issues facing Progger 3, and presents suggestions for future work to Progger 3 that could help to resolve these issues.

7.2 Comparison with similar work

7.2.1 Progger 1 and a TPM

The use of a TPM with Progger 1 has been explored by M. M. M. Bany Taha [3]. We will refer to this work hereafter as Progger 1 (TPM). There are similarities in objectives between Progger 3 and Progger 1 (TPM): both centre mostly around providing confidentiality and integrity of the collected provenance, but there are significant differences in scope.

Firstly, an important distinction is that Progger 1 (TPM) presents a framework design that combines a provenance client with other software, while Progger 3 is instead focused on providing a high-quality provenance client. We

consider many aspects of using Progger 3, such as the secure storage of the logs generated by Progger 3, and the route to achieving a trusted kernel, to be outside the scope of what Progger 3 should dictate, as there are many possible approaches depending on the circumstances. Meanwhile, Progger 1 (TPM) incorporates methods to store provenance logs with confidentiality, availability, and integrity, using a TPM to do so, while also detailing means to achieve a trusted execution environment.

Progger 3 uses a TPM for exactly one task: to store a cryptographic key so that the kernel may access it, but user space may not. When it comes to the client, Progger 1 (TPM) does not use the TPM for this at all, instead opting to require that data be transferred over TLS. Yet, as previously discussed, there is no full TLS implementation in the Linux kernel, so a user space program would be required. This thesis avoids evaluating Progger 1 (TPM) in terms of Progger 3's design goals, as the evaluation is mostly dependent on the underlying provenance client used. Yet, in this case, a requirement of Progger 1 (TPM) is to use a user space program. So, Progger 1 (TPM) would not satisfy design goal **A** of Progger 3 unless it was amended to utilise a provenance client like Progger 3 instead of using user space programs.

What Progger 1 (TPM) does use the TPM for on the client, apart from a trusted execution environment (including remote attestation), is maintaining the integrity of the data produced by Progger 1. Unlike Progger 3, Progger 1 (TPM) stores raw provenance records on the disk, as files on a regular file system. This leads to complexities around ensuring that any malicious modification of the files can be detected. However, in Progger 3, there is no need for the somewhat-complex client data integrity measurement that is presented in Progger 1 (TPM). This is because Progger 3 stores all of its data in kernel memory, which, by assumption, an attacker cannot manipulate when a trusted kernel is running. If kernel memory could be manipulated by an attacker, the attacker could modify the collected provenance before any confidentiality or integrity measures are applied. So, if an attacker could manipulate Progger 3's

data locally, Progger 1 (TPM) would also be vulnerable. Furthermore, if one were to extend Progger 3 to store data on-disk, it would not be necessary to implement extra integrity checks. This could be achieved by storing data on the disk only once it has had its confidentiality and integrity imposed by XChaCha20-Poly1305. Then, no additional special action needs to be taken to ensure the integrity of the data, as any manipulation will be immediately picked up when the data is decrypted. Either way, if Progger 3 is storing data on-disk or not, it can offer a much simpler environment for the client.

So, while the use of a TPM with Progger has been explored previously, it was for a different use case than Progger 3's use of a TPM. Progger 3 could be integrated with the framework proposed in Progger 1 (TPM), but is also able to be more versatile and can be integrated into environments with different requirements. Progger 3 provides a provenance client that can send the provenance it collects with confidentiality and integrity from a client running a trusted kernel to a remote server. Meanwhile, Progger 1 (TPM) provides a framework that utilises an existing provenance client, and where the scope extends to setting up a trusted execution environment on the client, securely storing the provenance logs, and providing remote attestation of the client.

7.2.2 Detecting commands executed as a different user

One of the four major breakthroughs Progger 1 lists is being able to determine when a non-root user executes a command with root privileges, logging the user that executed that command.

We have tested this. As Progger 1 was developed using CentOS 6.4 [2], the test used a virtual machine running the latest CentOS 6 series release: CentOS 6.10. Unfortunately, Progger 1 did not log the user running `sudo` as `user*`, as suggested by [2], instead printing `root`, demonstrated in Listing 7.1.

Listing 7.1: Progger 1 output

```
$ cat /secret-file
$ dmesg | grep '^Progger:0' | grep /secret-file
Progger:0,user,1948,1517,1517,1516,cat,/secret-file,/home/user/,0,1654284880,3
$ sudo cat /secret-file
$ dmesg | grep '^Progger:0' | grep /secret-file
Progger:0,user,1948,1517,1517,1516,cat,/secret-file,/home/user/,0,1654284880,3
Progger:0,root,1953,1952,1517,1516,cat,/secret-file,/home/user/,0,1452952576,3
```

This section does not seek to determine the issue in the source code responsible for the output in Listing 7.1. Instead, it analyses the ideas put forward in the paper presenting Progger 1 [2] for detecting users that execute commands as a different user. Then, it presents a way to detect this using Progger 3, which we consider more reliable than the approach in Progger 1.

To start the analysis, it is noted in [2] that, when a user executes `sudo`, some system calls are made as that user before the real and effective user IDs of the process are switched to the other user (usually 0, which is root). The example given in [2] is that `/etc/passwd` is opened by `sudo` before any user IDs are switched. It is further suggested by [2] that this can be used to detect when a user runs `sudo`. As we understand it, the suggestion here is that one could analyse the logs generated by Progger 1 to find entries containing a `comm` of `sudo`. From these entries, one might then look for `/etc/passwd` being opened before any user IDs are switched. Unfortunately, this can be circumvented, as shown in Listing 7.2.

Listing 7.2: Replacing the `comm` of `sudo`

```
$ ln -s /usr/bin/sudo arbitrary
$ ./arbitrary id
uid=0(root) gid=0(root) groups=0(root)
    context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
$ ./arbitrary
^Z
[1]+ Stopped                 ./arbitrary
$ pgrep arbitrary
1697
$ pgrep sudo
$ cat /proc/1697/comm
arbitrary
```

This shows that, by making a symlink to `sudo`, one can run `sudo` with the process's `comm` set arbitrarily. So, this method is not viable for detecting

when a user runs a command as a different user. Furthermore, the proposed method would have to account for any executables with the setuid bit set, not all of which will necessarily be named `sudo`.

The other idea presented in [2] to detect users running commands as other users is to look at session IDs. The idea is that the command executed by `sudo` (or any other privilege escalation program) will have the same session ID as the user that initiated `sudo`. We have created a simple program, `new-session`, whose code is in Appendix B, that shows that this idea does not always hold true. Listing 7.3 shows running `new-session` and its output.

Listing 7.3: Output of `new-session`

```
$ sudo ./new-session id
Orig. session ID: 502.
Orig. process ID: 1347.
  New session ID: 1347.
Child process ID: 1348.
  New session ID: 1348.
Child process ID: 1349.
execvp("id", ["id"])
uid=0(root) gid=0(root) groups=0(root)
```

What happens in Listing 7.3 is that `sudo` is used to gain privileges and execute the program `new-session`, which creates a new session. In this new session, the program is free to do as it wishes using the raised privileges. It creates multiple new sessions for a reason that is explained very shortly.

We ran `sudo strace -ff ./new-session` and found that, apart from system calls arising from printing debug information, the only system calls that occurred in the `main` function (which does not include the system calls made by the C library during program setup and exit) were `clone`, `setsid`, and `exit_group`. None of these system calls is traced by Progger 1. Additionally, `new-session` calls `setsid` and `fork` twice so that the final process's parent exits without performing any system calls traced by Progger 1. Hence, one also cannot look for the parent process ID or parent session ID to determine the user that ran `sudo`.

So, as we understand it, the methods proposed in [2] cannot be guaranteed

to always detect processes that are executed as a user different from the one that initiated the process. To be able to detect processes executed as a different user reliably, we have devised the following procedure. In the list of system calls being traced by Progger 3, include `fork`, `vfork`, `clone`, `clone3`, `setuid`, `setreuid`, and `setresuid`. Since Progger 3 provides the PID, UID, and EUID in its system call records, the server receiving the records can then keep a per-process list of changes in the (E)UID of a process. It could even just record the original (E)UID the first time any UID-changing system call is executed, for a simpler implementation. For any system call that occurs later, the server can check if that process was previously executing as a different user. If it was executing as a different user, one could find the first user it was executing as, and thus find the user that initiated the process. This must be repeated recursively for each parent of the process, to see if any ancestor was initially running as a different user. To see why, consider `sudo sudo <command>`. When determining the ancestors of a process, the server should look at the `fork` and `clone` system call records. This is because processes can change parents, but what is important is the process that initiated the process in question. The server can also use the `fork` and `clone` records to tell when a PID has been reused.

The ability to detect processes being executed as another user with this procedure is new in Progger 3, as previous iterations had no ability to trace `fork`, `vfork`, `clone`, `clone3`, `setuid`, `setreuid`, and `setresuid`.

7.3 Future work

Progger 3 has made advancements, but there are still issues that could be fixed or improved. The following sections will discuss these issues.

7.3.1 Utilising the disk to store buffers

Progger 3 stores all of its buffers that contain provenance in kernel memory. This is useful, as it means that data is easily protected from user space access, but it is a limited resource. Progger 3 can configure the amount of memory to use for buffers, and around 512 KiB per CPU has been found to be sufficient when tracing only `openat` and having a fast network connection to the Progger 3 server. But, if a large number of traced system calls occur in a short space of time, or the network connection is unstable, the buffers could overflow, losing information. In this scenario, the larger amount of storage available from disks could be utilised to avoid losing data. Having data stored on-disk provides a new avenue by which user space could potentially tamper with the provenance. Yet, since Progger 3 protects its buffers with XChaCha20-Poly1305, one wouldn't need to take extra care in terms of data confidentiality and integrity, as any tampering could be detected during decryption. Still, care would need to be taken to ensure that user space could not delete the data stored on the disk by Progger 3.

7.3.2 User space network interface control

In most, if not all Linux-based systems, user space is in control of bringing network interfaces up and down. This means that user space could bring the required network interface down to stop Progger 3 from being able to send provenance. During this time, Progger 3 is still collecting the provenance and will send it when the connection is restored. However, if the interface is down for long enough, Progger 3 could run out of room in its ring buffer to store more data, and provenance would be lost. At least, one may be able to detect a network interface being disconnected by observing a cessation of system call records being received by the server without the system call records indicating a shutdown took place.

In addition to user space being able to control the network interfaces maliciously, there is also the issue that, when using Progger 3 as a built-in module,

as intended, there is a delay between when system call tracing starts and when the network interfaces have been brought up by user space. This delay means that, without reasonably large buffers, Progger 3 is prone to experiencing buffer overflows early on in the system startup sequence.

This issue could be mitigated to some extent by extending Progger 3 to store buffers on the disk(s), as we have previously suggested, but it would not truly fix the problem. User space could disconnect the interface indefinitely, or just long enough for even the disk space allocated for buffers to run out. One must also consider that the physical network interface, such as an Ethernet cable, can be disconnected, or an attacker in the network path could drop packets, in order to achieve a similar effect.

7.3.3 Task priority

One aspect we considered when designing Progger 3 is that tasks running in the kernel still have to compete with user space tasks for scheduling priority. If many high-priority tasks were spawned in user space, and Progger 3's tasks were not at a sufficiently high priority, Progger 3 may only get a very small amount of time to run. By design, using tracepoints, Progger 3 will always be able to record each system call, regardless of the priority at which Progger 3's tasks are running. However, the issue lies in the sending of the data Progger 3 collects. If the tasks Progger 3 spawns do not have a chance to run often enough to empty the ring buffer of system call records, by sending these records over TCP to the server, ring buffer overflows may occur. In other words, records about system calls may be lost.

In order to make such an event less likely, Progger 3 sets the priority of its tasks to the maximum possible. This does carry some risk, as an errant Progger 3 task caught in an infinite loop could completely consume a CPU core. However, we are confident enough in Progger 3's code quality to implement this.

This measure protects against systems that genuinely may be experiencing

high workloads with high-priority tasks. Unfortunately, a malicious actor can easily lower the priority of kernel tasks using `chrt`, and then create their own tasks with higher priorities. When it comes to preventing malicious processes raising their priority to deny other tasks CPU time, Linux leaves that task to the administrator in user space (using `setrlimit` and `cgroups`, for example), to the best of our understanding.

One might be able to resolve this issue by making user space unable to change the priority of kernel tasks, and ensuring the maximum priority of a kernel task is greater than that of a user space task (at least for the `SCHED_FIFO` scheduling policy). We consider this to be absolutely outside the scope of Progger 3.

So, through creating very high priority tasks, a malicious user space could affect the availability of system call records generated by Progger 3.

It is at least worth pointing out that this could be fixed by having Progger 3 send the system call record to the remote server while still in the tracepoint. However, Progger 3 has been designed explicitly not to do this, and use a ring buffer instead, so that the performance impact is minimal. Any action made in a system call tracepoint blocks that system call entirely. Sending data over TCP could take time on the order of milliseconds to complete, compared to microseconds for a typical system call.

7.3.4 Copying system call pointer arguments

As noted previously, Progger 3 opts to copy only integer and C-string arguments of system calls. This is primarily to reduce code complexity while still providing the most pertinent information for many use cases. However, should one desire, judging the increased code complexity to be worthwhile, Progger 3 could be modified to copy all system call arguments, including pointer values. These pointers may point to complicated structures, so it would take a lot of care to do correctly.

7.3.5 Reducing bandwidth usage

As discussed in subsection 6.1.2, Progger 3 can use quite a lot of network bandwidth. Being able to configure Progger 3 to send data on system calls where some predicate is matched, such as paths for `openat` being in a specific directory, has potential to reduce this bandwidth usage considerably. However, it would also come at the cost of increased code complexity, and could cause at least a slight performance hit.

7.3.6 Information leakage

On a system running Progger 3, activity can be inferred by observing network activity. If Progger 3 is monitoring file accesses, opening a file will almost immediately cause a network packet to be sent. The details of the action will be protected from an attacker, but that an action simply occurred can potentially be useful to an attacker. One potential case is that an attacker could determine times that a system is in use, and times when it is not, with higher accuracy than if Progger 3 were not running. This may give the attacker a better idea of suitable times to carry out other attacks. Actions could be taken, such as batching messages, to reduce the impact of this information leakage.

One might also consider periodically rotating the client ID and fixed part of the XChaCha20-Poly1305 nonce so that it is harder for an observer to match network traffic to individual systems using Progger 3.

7.3.7 Namespaces

With each system call record collected by Progger 3, the PIDs and UIDs are relative to their initial namespaces. So, for example, PID 1 inside a container will be logged with its PID as viewed from outside the container. It is simple to add support for namespace-relative PIDs and UIDs, but it wasn't included in the end as it does make each system call take a reasonable amount of time

longer to complete. Should there be a case where this tradeoff is worthwhile, one may wish to implement such support.

7.3.8 The Progger 3 server

Chapter 4 discusses our implementation of a server to receive the data generated by the Progger 3 client. We consider our implementation to be more than adequate for the purposes of testing and debugging the Progger 3 client. The Progger 3 client is by far the most important component of the two: there could be many different servers for Progger 3, each with their own goals, but the design and goals of the Progger 3 client remain constant. As such, we wanted to perfect the Progger 3 client, but a server with some minor issues remained consistent with our goals.

As a result, while our server implementation is quite usable, there are some minor bugs and unimplemented security features, such as checking the sequence number of messages (embedded in the nonce). If one desires, there is work that could be done to create a more secure and widely-useful server for Progger 3 that may be suitable for real-world deployments.

7.3.9 Choosing which system calls to trace

For comprehensive provenance collection, it can be challenging to determine exactly which system calls need to be traced. For opening a file, there are three system calls: `open`, `openat`, and `openat2`. There is a multitude of system calls relating to file writing, such as `write`, `pwrite`, and `lseek`, that would all need to be traced to get a complete picture. Furthermore, a file can be memory-mapped with with `mmap` and then written to as a standard memory object without using system calls.

In contrast, the approach taken in CamFlow [17] of using a Linux Security Module (LSM) is "guaranteed to capture every event that is deemed security-sensitive and focus on the objects being accessed, instead of the actions being carried out on those objects." This approach could reduce the complexities of

selecting and processing the correct system calls.

As such, one may wish to explore implementing Progger 3 as an LSM, seeing if the design goals of Progger 3 (listed in section 1.1) can still be met.

7.4 Summary

This chapter has explored some ways in which Progger 3 differs from similar work, which may not have been apparent from earlier chapters. It should now be clear how the existing framework that combines Progger 1 with a TPM [3] differs from Progger 3; particularly, the framework does not provide a provenance client implementation. Compared to this framework, Progger 3 is able to be used in a wider variety of environments, owing to its more focused scope. Additionally, Progger 3 is better equipped to deal with detecting commands being executed as different users, compared to previous versions of Progger.

This chapter has also covered some issues we have identified in Progger 3, and made suggestions for how one may approach fixing these issues. For all of these issues, there was a reason that we did not resolve the issue in Progger 3: either that the issue is outside the scope of Progger 3 (subsections 7.3.2 and 7.3.3); or fixing the issue involves trade-offs, such as increased complexity (subsections 7.3.1, 7.3.4, 7.3.5, and 7.3.6) or reduced performance (subsection 7.3.7); or that the issue is not one to be fixed per se, but rather a project in its own right for future exploration (subsections 7.3.8 and 7.3.9). Although these issues exist, they do not stop Progger 3 from achieving its design goals.

Chapter 8

Conclusion

This thesis has presented Progger 3, a provenance system that traces Linux system calls in order to collect provenance of data. Progger 3 has made many advancements relative to its predecessors: Progger 1 and Progger 2. In particular, Progger 3 is vastly more performant, to the point where we consider it the first iteration of Progger that can be realistically used in many real-world workloads without causing an unacceptable loss in performance. In addition to performance, Progger 3 offers a new level in security, being able to ensure the collected provenance cannot be tampered with, so long as the kernel on the client is not compromised. The provenance is tamper-proof even in the event of a compromised user space on the client. Furthermore, Progger 3 is a working implementation of our ideas, and it shows excellent stability and ease of use. Finally, Progger 3 is able to trace any system call, compared to the relatively small subsets supported in Progger 1 and Progger 2. We believe that, should one desire to use a secure, efficient data provenance system that traces system calls, Progger 3 will likely meet their needs better than any other existing system.

References

[1] L. Carata *et al.*, "A primer on provenance," *Communications of the ACM*, vol. 57, no. 5, pp. 52–60, 2014.

[2] R. K. L. Ko and M. A. Will, "Progger: An efficient, tamper-evident kernel-space logger for cloud data provenance tracking," in *IEEE 7th International Conference on Cloud Computing*, 2014, pp. 881–889. DOI: 10.1109/CLOUD.2014.121.

[3] M. M. M. Bany Taha, "Tamper-evident data provenance," Master of Engineering thesis, University of Waikato, Hamilton, New Zealand, 2016. [Online]. Available: https://hdl.handle.net/10289/9972.

[4] L. Torvalds *et al.*, *64-bit system call numbers and entry vectors*, version 5.8.18, May 14, 2020. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/entry/syscalls/syscall_64.tbl?h=v5.8.18.

[5] Trusted Computing Group, *Trusted Platform Module Library*, Family "2.0", Rev. 01.59, Nov. 8, 2019. [Online]. Available: https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[6] J. Yao and V. J. Zimmer, "Intel® Boot Guard," in *Understanding the UEFI Secure Boot Chain*. Jun. 2019. [Online]. Available: https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/secure_boot_chain_in_uefi/intel_boot_guard.

[7] W. Liu and S. Jones, "Cyber-resiliency in chipset and BIOS," Tech. Rep., 2017. [Online]. Available: https://downloads.dell.com/solutions/

servers-solution-resources/Direct%20from%20Development%20-%20Cyber-Resiliency%20In%20Chipset%20and%20BIOS.pdf.

[8] Intel Corporation, *Intel® 8 Series/C220 Series Chipset Family Platform Controller Hub (PCH)*, May 2014, pp. 729–732. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/8-series-chipset-pch-datasheet.pdf.

[9] Winbond Electronics Corporation, *W25Q64FV*, Oct. 26, 2016. [Online]. Available: https://www.winbond.com/resource-files/w25q64fv%20revr%2010262016.pdf.

[10] L. Torvalds *et al.*, *The kernel's command-line parameters*, version 5.8.18, Sep. 7, 2020. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/admin-guide/kernel-parameters.txt?h=v5.8.18.

[11] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF protocols," RFC 8439, Jun. 2018. [Online]. Available: https://tools.ietf.org/html/rfc8439.

[12] S. Arciszewski, "XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305, draft-irtf-cfrg-xchacha-03," Tech. Rep., Jan. 10, 2020. [Online]. Available: https://tools.ietf.org/html/draft-irtf-cfrg-xchacha-03.

[13] B. Pérez, J. Rubio, and C. Sáenz-Adán, "A systematic review of provenance systems," *Knowledge and Information Systems*, vol. 57, no. 3, pp. 495–543, 2018.

[14] Red Hat, Inc., "Understanding the eBPF networking features in RHEL," in *Configuring and Managing Networking, A guide to configuring and managing networking in Red Hat Enterprise Linux 8*. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/assembly_understanding-the-ebpf-features-in-rhel_configuring-and-managing-networking.

[15] K.-K. Muniswamy-Reddy, D. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proceedings of the USENIX 2006 Annual Technical Conference*, Jan. 2006, pp. 43–56.

[16] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer, "Provenance for the cloud," in *Proceedings of the 8th Conference on File and Storage Technologies (FAST'10)*, Feb. 2010.

[17] T. Pasquier *et al.*, "Practical whole-system provenance capture," in *Symposium on Cloud Computing (SoCC'17)*, ACM, 2017.

[18] L. Torvalds *et al.*, *Linux Security Modules: General security hooks for Linux*, version 5.8.18, Apr. 21, 2020. [Online]. Available: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/security/lsm.rst?h=v5.8.18`.

[19] "Sysdig," sysdig.com. [Online]. Available: `https://sysdig.com/` (accessed Nov. 16, 2020).

[20] "SystemTap," sourceware.org. [Online]. Available: `https://sourceware.org/systemtap/index.html` (accessed Nov. 16, 2020).

[21] "bpftrace," bpftrace.org. [Online]. Available: `https://bpftrace.org/` (accessed Nov. 16, 2020).

[22] "What is eBPF?" ebpf.io. [Online]. Available: `https://ebpf.io/what-is-ebpf/` (accessed Dec. 10, 2020).

[23] L. Torvalds *et al.*, *Kernel TLS*, version 5.8.18, Oct. 4, 2019. [Online]. Available: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/networking/tls.rst?h=v5.8.18`.

[24] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," RFC 8446, Aug. 2018. [Online]. Available: `https://tools.ietf.org/html/rfc8446`.

[25] W. Roberts *et al.*, *tpm2-tools*, version 4.2, Apr. 8, 2020. [Online]. Available: https://github.com/tpm2-software/tpm2-tools.

[26] Trusted Computing Group, *TCG PC client platform TPM profile specification for TPM 2.0*, version 1.05, revision 14, Sep. 4, 2020. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/PC-Client-Specific-Platform-TPM-Profile-for-TPM-2p0-v1p05p_r14_pub.pdf.

[27] D. Khovratovich, C. Rechberger, and A. Savelieva, *Bicliques for preimages: Attacks on Skein-512 and the SHA-2 family*, Cryptology ePrint Archive, Report 2011/286, 2011. [Online]. Available: https://eprint.iacr.org/2011/286.

[28] S. Berger *et al.*, *swtpm*, version 0.3.1, Mar. 30, 2020. [Online]. Available: https://github.com/stefanberger/swtpm.

[29] F. Bellard *et al.*, *QEMU*, version 3.1, Dec. 11, 2018. [Online]. Available: https://www.qemu.org/.

Appendices

Appendix A

Progger 3 source code

These files are placed in the Linux source tree. Development was done against Linux 5.8.y, but it would be straightforward to rebase to a later version.

Progger 3 is licensed under the GPLv2.

A patch file for this code against Linux 5.8.y is attached to this PDF for easier access if you want to use the source code. Right-click and save *this attachment* to obtain it.

A.1 Kconfig (diff)

```
diff --git a/Kconfig b/Kconfig
index 745bc773f567..f3ad6ff1ad80 100644
--- a/Kconfig
+++ b/Kconfig
@@ -7,6 +7,9 @@ mainmenu "Linux/$(ARCH) $(KERNELVERSION) Kernel Configuration"

 source "scripts/Kconfig.include"

+# Only placed near the top for development purposes.
+source "drivers/net/progger/Kconfig"
+
 source "init/Kconfig"

 source "kernel/Kconfig.freezer"
```

A.2 drivers/net/Makefile (diff)

```
diff --git a/drivers/net/Makefile b/drivers/net/Makefile
index 94b60800887a..887b8ca2cfb0 100644
--- a/drivers/net/Makefile
```

```
+++ b/drivers/net/Makefile
@@ -82,3 +82,6 @@ thunderbolt-net-y += thunderbolt.o
 obj-$(CONFIG_USB4_NET) += thunderbolt-net.o
 obj-$(CONFIG_NETDEVSIM) += netdevsim/
 obj-$(CONFIG_NET_FAILOVER) += net_failover.o
+
+obj-$(CONFIG_PROGGER) += progger/
+subdir-$(CONFIG_PROGGER) += progger
```

A.3 drivers/net/progger/Kconfig

```
# SPDX-License-Identifier: GPL-2.0-only

menu "Progger configuration"

config PROGGER
	tristate "Progger support"
	depends on X86
	depends on 64BIT
	depends on NET && INET
	select CRYPTO
	select CRYPTO_LIB_CHACHA20POLY1305
	select CRYPTO_CHACHA20_X86_64
	select CRYPTO_POLY1305_X86_64
	select TRACING

if PROGGER

source "drivers/net/progger/kernel/Kconfig"

config PROGGER_CLIENT_ID
	hex "Client ID"
	default 0xcafe1337

endif # Progger

endmenu # Progger configuration
```

A.4 drivers/net/progger/Makefile

```
# SPDX-License-Identifier: GPL-2.0-only

subdir-$(CONFIG_PROGGER) += kernel
subdir-$(CONFIG_PROGGER) += server

obj-$(CONFIG_PROGGER) += kernel/
obj-$(CONFIG_PROGGER) += server/
```

A.5 drivers/net/progger/kernel/Kconfig

```
# SPDX-License-Identifier: GPL-2.0-only

```

```
3  menu "Tracepoints"
4
5  #
6  # Each tracepoint config must be of the form PROGGER_TRACEPOINT_x, where x is
7  # the name of an available tracepoint. This is so the resulting config can be
8  # parsed to automatically generate code listing the tracepoints to use.
9  #
10
11 config PROGGER_TRACE_SYSCALLS
12        bool "Trace syscalls"
13        default y
14        select PROGGER_TRACEPOINT_SYS_ENTER
15        select PROGGER_TRACEPOINT_SYS_EXIT
16
17 config PROGGER_TRACEPOINT_SYS_ENTER
18        bool
19
20 config PROGGER_TRACEPOINT_SYS_EXIT
21        bool
22
23 config PROGGER_TRACED_SYSCALLS
24        string "System calls to trace"
25        depends on PROGGER_TRACE_SYSCALLS
26        help
27          A regular expression that matches the system calls to be traced.
28
29          For example:
30           - "openat|rename(at)?2?|u?mount2?"
31           - ".*"
32           - ".*xattr.*"
33
34 endmenu # Tracepoints
35
36 config PROGGER_USE_TPM
37        bool "Use the TPM"
38        default y
39        select TCG_TPM
40        select TCG_TIS
41        help
42          For development or testing, it may be preferable to use a
43          system without a TPM. Only say N if that is the case.
44
45 if PROGGER_USE_TPM
46
47 config PROGGER_TPM_PCR
48        int "PCR to allocate to Progger"
49        range 8 15
50        default 15
51
52 config PROGGER_TPM_KEY_HANDLE
53        hex "Parent key handle"
54        help
55          The handle of the key used to seal the ChaCha20-Poly1305 key.
56          It is not the handle of the ChaCha20-Poly1305 key.
57
58 config PROGGER_TPM_PUBLIC_BLOB
59        string "Public blob path"
60        help
61          The path to the public blob generated while sealing the crypto key.
62
```

```
63  config PROGGER_TPM_PRIVATE_BLOB
64          string "Private blob path"
65          help
66            The path to the private blob generated while sealing the crypto key.
67
68  config PROGGER_PANIC_WHEN_KEY_UNSECURED
69          bool "Panic when the sealed crypto key is left unsecured"
70          default y
71          help
72            The crypto key can only be unsealed once per boot. If an error occurs
73            when the crypto key is being unsealed, user space could potentially
74            unseal the key later. The only option left to protect the crypto key
75            from user space is to force a kernel panic.
76
77            This should only be set to N for development or testing.
78
79  config PROGGER_PANIC_WHEN_NO_TPM_FOUND
80          bool "Panic when no TPM device can be found"
81          depends on PROGGER_PANIC_WHEN_KEY_UNSECURED
82          default y
83          help
84            To be extra sure that the crypto key is not accessible to user space,
85            Progger can panic if it doesn't find a TPM.
86
87            Suppose Progger is compiled built-in, and the TPM driver is compiled
88            only as a loadable module. In this case, Progger will not find the
89            TPM during initialisation, but the TPM will become available to
90            the system when the TPM module is loaded from user space. As
91            such, Progger is unable to unseal the crypto key and secure it
92            from user space.
93
94            If you have ensured that the relevant TPM driver is compiled
95            built-in, then this option is not needed. In such a case, it
96            could be beneficial to disable this option, as that would mean
97            that system is still bootable if the TPM hardware fails.
98
99  config PROGGER_PANIC_WHEN_TPM_IS_NOT_VERSION_2
100         bool "Panic when TPM device is not a TPM 2.0 device"
101         depends on PROGGER_PANIC_WHEN_KEY_UNSECURED
102         default y
103         help
104           Progger requires a device supporting TPM 2.0. If this is not the
105           case, Progger cannot protect the sealed crypto key from user space.
106           The only option left to protect the crypto key from user space is to
107           panic the kernel.
108
109           If this option is not set, and Progger encounters a device that
110           does not support TPM 2.0, Progger won't try to unseal the key;
111           instead, Progger will just fail its initialisation.
112
113 endif # PROGGER_USR_TPM
114
115 config PROGGER_NET_DSTADDR
116         string "Destination IP address"
117         default "::1"
118         help
119           The IP address to send records to. Both IPv4 and IPv6 are supported.
120
121 config PROGGER_RINGBUF_SIZE
122         int "Record ringbuffer size (KiB per CPU)"
```

```
123         default 256
124         help
125           The number of kilobytes per CPU to use as a buffer for records.
126           Large buffers might be required to prevent data loss when a lot
127           of traced events occur in a small timeframe.
128
129           If Progger is compiled built-in instead of as a module, there will be
130           a period where Progger is collecting records but is unable to send
131           them, as the network interfaces are not yet up. The ringbuffer size
132           can be increased to compensate for that.
```

A.6 drivers/net/progger/kernel/Makefile

```
1  # SPDX-License-Identifier: GPL-2.0-only
2
3  obj-$(CONFIG_PROGGER) := progger.o
4
5  progger-y := init.o
6  progger-y += tracepoints.o
7  progger-y += crypto.o
8  progger-y += net.o
9  progger-y += ringbuf.o
10 progger-y += kthread.o
11 progger-$(CONFIG_PROGGER_USE_TPM) += tpm.o
12
13 ccflags-y += -O3
14 ccflags-y += -D'pr_fmt(fmt)=KBUILD_MODNAME ": " fmt'
15
16 syscall-tbl := $(srctree)/arch/x86/entry/syscalls/syscall_64.tbl
17
18 generated := $(obj)/generated
19
20 tpm-blobs := tpm-public-blob.h
21 tpm-blobs += tpm-private-blob.h
22
23 clean-files += generated/ip.h
24 clean-files += generated/syscalls.h
25 clean-files += $(addprefix generated/,$(tpm-blobs))
26 clean-files += $(addprefix generated/,$(tpm-blobs).tmp)
27
28 quiet_cmd_progger_gen = GEN  $@
29      cmd_progger_gen = $< $(KCONFIG_CONFIG) $@ $(2)
30
31 quiet_cmd_tpm_blob_gen = GEN  $@
32      cmd_tpm_blob_gen = \
33               xxd -i >$@.tmp <$(CONFIG_PROGGER_TPM_$(2)_BLOB) && \
34               if ! diff -N $@ $@.tmp >/dev/null; then mv $@.tmp $@; fi
35
36 $(obj)/net.o: $(generated)/ip.h
37 $(obj)/tracepoints.o: $(generated)/syscalls.h
38 $(obj)/tpm.o: $(addprefix $(generated)/,$(tpm-blobs))
39
40 $(generated)/ip.h: $(generated)/gen-ip.py
41       $(call cmd,progger_gen)
42
43 $(generated)/syscalls.h: $(generated)/gen-syscalls.py $(syscall-tbl)
44       $(call cmd,progger_gen,$(syscall-tbl))
```

```
45
46 $(generated)/tpm-public-blob.h: FORCE
47         $(call cmd,tpm_blob_gen,PUBLIC)
48
49 $(generated)/tpm-private-blob.h: FORCE
50         $(call cmd,tpm_blob_gen,PRIVATE)
51
52 $(wildcard $(generated)/*.h): $(KCONFIG_CONFIG)
53
54 $(wildcard $(generated)/gen-*.py): ;
55
56 FORCE: ;
```

A.7 drivers/net/progger/kernel/crypto.c

```
1  // SPDX-License-Identifier: GPL-2.0-only
2
3  #include "crypto.h"
4
5  #include <crypto/chacha20poly1305.h>
6  #include <linux/cache.h>
7  #include <linux/compiler.h>
8  #include <linux/errno.h>
9  #include <linux/init.h>
10 #include <linux/kconfig.h>
11 #include <linux/kernel.h>
12 #include <linux/limits.h>
13 #include <linux/mm.h>
14 #include <linux/overflow.h>
15 #include <linux/random.h>
16 #include <linux/spinlock.h>
17 #include <linux/string.h>
18
19 #include <progger/crypto.h>
20 #include <progger/types.h>
21
22 #include "tpm.h"
23
24 static u8 __read_mostly key[CHACHA20POLY1305_KEY_SIZE];
25 static u8 nonce[XCHACHA20POLY1305_NONCE_SIZE];
26 static DEFINE_SPINLOCK(nonce_lock);
27 static u64 counter;
28
29 int progger_xchacha20poly1305(void **out, size_t *outlen,
30                               void *data, size_t datalen,
31                               void *ad, size_t adlen)
32 {
33         size_t extralen = CHACHA20POLY1305_AUTHTAG_SIZE + sizeof(nonce);
34
35         if (unlikely(counter == U64_MAX)) {
36                 pr_err_once("Nonce counter has reached its maximum!");
37                 return -EOVERFLOW;
38         }
39
40         if (unlikely(check_add_overflow(datalen, extralen, outlen)))
41                 return -EOVERFLOW;
42
```

```
43         *out = kvmalloc(*outlen, GFP_KERNEL);
44         if (unlikely(!*out))
45                 return -ENOMEM;
46
47         spin_lock(&nonce_lock);
48         *((u64 *)nonce + 2) = counter++;
49         memcpy(*out, nonce, sizeof(nonce));
50         spin_unlock(&nonce_lock);
51
52         xchacha20poly1305_encrypt(*out + sizeof(nonce), data, datalen,
53                                   ad, adlen, *out, key);
54
55         return 0;
56 }
57
58 int __init crypto_init(void)
59 {
60         int err;
61
62         if (IS_ENABLED(CONFIG_PROGGER_USE_TPM)) {
63                 err = tpm_get_chacha20poly1305_key(&key);
64                 if (err)
65                         return err;
66         } else {
67                 pr_warn("Not using TPM, using insecure testing key.\n");
68                 memcpy(key, progger_crypto_testkey, sizeof(key));
69         }
70
71         err = get_random_bytes_wait(&nonce, sizeof(nonce));
72         if (err)
73                 return err;
74
75         return 0;
76 }
77
78 void crypto_exit(void)
79 {
80         memset(key, 0, sizeof(key));
81 }
```

A.8 drivers/net/progger/kernel/crypto.h

```
1  /* SPDX-License-Identifier: GPL-2.0-only */
2
3  #ifndef PROGGER_CRYPTO_H
4  #define PROGGER_CRYPTO_H
5
6  #include <progger/types.h>
7
8  int progger_xchacha20poly1305(void **out, size_t *outlen,
9                                void *data, size_t datalen,
10                               void *ad, size_t adlen);
11 int crypto_init(void);
12 void crypto_exit(void);
13
14 #endif /* PROGGER_CRYPTO_H */
```

A.9 drivers/net/progger/kernel/generated/gen-ip.py

```
#!/usr/bin/env python3
# SPDX-License-Identifier: GPL-2.0-only

import ipaddress
import sys


def main():
    if len(sys.argv) != 3:
        print(f"Usage: {sys.argv[0]} <.config> <output-file>", file=sys.stderr)
        exit(1)

    dot_config = sys.argv[1]
    output_file = sys.argv[2]
    output_lines = []

    with open(dot_config, 'r') as f:
        for line in f.readlines():
            if line.startswith('CONFIG_PROGGER_NET_DSTADDR='):
                addr = line.split('=')[-1].strip().replace('"', '')
                addr = ipaddress.ip_address(addr)

                formatted = None

                if type(addr) == ipaddress.IPv4Address:
                    v4addr = f'{int(addr)} /* {addr} */'
                    v6addr = '{ 0, } /* None */'
                    output_lines.append('#define PROGGER_IPv6 0')
                else:
                    v4addr = '0 /* None */'
                    v6addr = ','.join([f'0x{byte:02x}' for byte in addr.packed])
                    v6addr = f'{{ {v6addr} }} /* {addr} */'
                    output_lines.append('#define PROGGER_IPv6 1')

                output_lines.append((f'#define PROGGER_IPv4_DSTADDR {v4addr}'))
                output_lines.append((f'#define PROGGER_IPv6_DSTADDR {v6addr}'))

    output = '\n'.join(output_lines) + '\n'

    with open(output_file, 'w') as f:
        f.write(output)


if __name__ == '__main__':
    main()
```

A.10 drivers/net/progger/kernel/generated/gen-syscalls.py

```
#!/usr/bin/env python3
# SPDX-License-Identifier: GPL-2.0-only

import re
import sys

TRACEPOINT_CONFIG_START = 'CONFIG_PROGGER_TRACEPOINT_'


def format_c_array(array_name, member_type, contents):
    newline = '\n'
    tab = '\t'

    if not member_type.endswith('*'):
        member_type += ' '

    return f'''\
static {member_type}{array_name}[] = {{
{tab}{(',' + newline + tab).join(contents)}
}};\
'''


def main():
    if len(sys.argv) != 4:
        print(f"Usage: {sys.argv[0]} <.config> <output-file> <syscall-tbl>",
              file=sys.stderr)
        exit(1)

    dot_config = sys.argv[1]
    output_file = sys.argv[2]
    syscall_tbl = sys.argv[3]
    tracepoints = []
    traced_syscalls = []
    syscall_regex = ''

    with open(dot_config, 'r') as f:
        for line in f.readlines():
            if line.startswith(TRACEPOINT_CONFIG_START):
                line = line[len(TRACEPOINT_CONFIG_START):]
                tracepoint = (line.split('=')[0].strip().lower())
                tracepoints.append(f'{{ {tracepoint}_tp, "{tracepoint}" }}')

            if line.startswith('CONFIG_PROGGER_TRACED_SYSCALLS'):
                syscall_regex = line.split('=', 1)[1].strip()[1:-1]

    syscall_regex = re.compile(syscall_regex)

    with open(syscall_tbl, 'r') as f:
        for line in f.readlines():
            line = line.strip()

            if not line or line.startswith('#'):
                continue

            parts = line.split()
            name = parts[2]

            if syscall_regex.fullmatch(name):
                traced_syscalls.append(f'__NR_{name}')
```

```
61
62
63      tp_type = '''\
64  const struct {
65          void *fn;
66          const char *name;
67  }'''
68
69      output = '\n'.join((
70          format_c_array('init_tracepoints', tp_type, tracepoints), '',
71          format_c_array('init_syscalls', 'const long', traced_syscalls), '',
72      ))
73
74      with open(output_file, 'w') as f:
75          f.write(output)
76
77
78  if __name__ == '__main__':
79      main()
```

A.11 drivers/net/progger/kernel/init.c

```
1   // SPDX-License-Identifier: GPL-2.0-only
2
3   #include <linux/init.h>
4   #include <linux/module.h>
5   #include <linux/printk.h>
6
7   #include "crypto.h"
8   #include "net.h"
9   #include "ringbuf.h"
10  #include "tpm.h"
11  #include "tracepoints.h"
12
13  static int __init progger_init(void)
14  {
15          int err;
16
17          err = tpm_init();
18          if (err) {
19                  pr_err("TPM init failed!\n");
20                  goto fail_tpm_init;
21          }
22
23          err = crypto_init();
24          if (err) {
25                  pr_err("Crypto init failed!\n");
26                  goto fail_crypto_init;
27          }
28
29          err = net_init();
30          if (err) {
31                  pr_err("Net init failed!\n");
32                  goto fail_net;
33          }
34
35          err = ringbuf_init();
```

```
36          if (err) {
37                  pr_err("Ringbuf init failed!\n");
38                  goto fail_ringbuf;
39          }
40
41          err = tracepoint_init();
42          if (err) {
43                  pr_err("Tracepoint init failed!\n");
44                  goto fail_tracepoints;
45          }
46
47          return 0;
48
49  fail_tracepoints:
50          tracepoint_exit();
51  fail_ringbuf:
52          ringbuf_exit();
53  fail_net:
54          net_exit();
55  fail_crypto_init:
56          crypto_exit();
57  fail_tpm_init:
58          tpm_exit();
59
60          return err;
61  }
62
63  static void __exit progger_exit(void)
64  {
65          tracepoint_exit();
66          ringbuf_exit();
67          net_exit();
68          crypto_exit();
69          tpm_exit();
70  }
71
72  /*
73   * We need to use ‘late_initcall‘ so that the TPM driver is initialised
74   * before ‘progger_init‘ runs. It’s fine for ‘progger_init‘ to run earlier
75   * when the TPM isn’t being used, but there’s also no need as there will be
76   * nothing for Progger to trace until user space starts.
77   */
78  late_initcall(progger_init);
79  module_exit(progger_exit);
80
81  MODULE_LICENSE("GPL v2");
```

A.12 drivers/net/progger/kernel/kthread.c

```
1  // SPDX-License-Identifier: GPL-2.0-only
2
3  #include "kthread.h"
4
5  #include <linux/err.h>
6  #include <linux/kthread.h>
7  #include <linux/sched.h>
8  #include <linux/sched/prio.h>
```

```
9  #include <linux/sched/task.h>
10 #include <linux/stddef.h>
11 #include <uapi/linux/sched/types.h>
12
13 int progger_create_task(struct task_struct **task, int (*threadfn)(void *data),
14                         const char *name)
15 {
16         struct task_struct *new;
17         const struct sched_param sp = { .sched_priority = MAX_RT_PRIO - 1 };
18
19         new = kthread_run(threadfn, NULL, name);
20         if (IS_ERR(new))
21                 return PTR_ERR(new);
22
23         sched_setscheduler(new, SCHED_FIFO, &sp);
24
25         *task = get_task_struct(new);
26
27         return 0;
28 }
29
30 void progger_destroy_task(struct task_struct **task)
31 {
32         if (!*task)
33                 return;
34
35         kthread_park(*task);
36         kthread_stop(*task);
37         put_task_struct(*task);
38
39         *task = NULL;
40 }
```

A.13 drivers/net/progger/kernel/kthread.h

```
1  /* SPDX-License-Identifier: GPL-2.0-only */
2
3  #ifndef PROGGER_KERNEL_KTHREAD_H
4  #define PROGGER_KERNEL_KTHREAD_H
5
6  #include <linux/sched.h>
7
8  int progger_create_task(struct task_struct **task, int (*threadfn)(void *data),
9                          const char *name);
10 void progger_destroy_task(struct task_struct **task);
11
12 #endif /* PROGGER_KERNEL_KTHREAD_H */
```

A.14 drivers/net/progger/kernel/net.c

```
1  // SPDX-License-Identifier: GPL-2.0
2
3  #include "net.h"
4
```

```
5  #include <crypto/chacha20poly1305.h>
6  #include <linux/byteorder/generic.h>
7  #include <linux/err.h>
8  #include <linux/errno.h>
9  #include <linux/in6.h>
10 #include <linux/in.h>
11 #include <linux/init.h>
12 #include <linux/jiffies.h>
13 #include <linux/kthread.h>
14 #include <linux/limits.h>
15 #include <linux/mm.h>
16 #include <linux/net.h>
17 #include <linux/printk.h>
18 #include <linux/sched.h>
19 #include <linux/socket.h>
20 #include <linux/uio.h>
21 #include <net/ipv6.h>
22 #include <net/net_namespace.h>
23 #include <net/sock.h>
24
25 #include <progger/net.h>
26 #include <progger/record.h>
27 #include <progger/types.h>
28
29 #include "crypto.h"
30 #include "kthread.h"
31 #include "generated/ip.h"
32
33 static struct net *net = &init_net;
34 static struct socket *sock;
35 static bool sock_ready;
36
37 static struct task_struct *tcp_worker;
38
39 static void progger_release_sock(void)
40 {
41         if (!sock)
42                 return;
43
44         sock_ready = 0;
45         kernel_sock_shutdown(sock, SHUT_WR);
46         sock_release(sock);
47         sock = NULL;
48 }
49
50 static int progger_connect(void)
51 {
52         int err;
53         int dstaddrlen;
54         struct sockaddr *dstaddr;
55         const __kernel_sa_family_t family = PROGGER_IPv6 ? AF_INET6 : AF_INET;
56
57         static struct sockaddr_in6 dstaddr6 = {
58                 .sin6_family = AF_INET6,
59                 .sin6_addr = { .s6_addr = PROGGER_IPv6_DSTADDR },
60                 .sin6_port = htons(PROGGER_SERVER_PORT),
61         };
62
63         static const struct sockaddr_in dstaddr4 = {
64                 .sin_family = AF_INET,
```

```
65                .sin_addr = { htonl(PROGGER_IPv4_DSTADDR) },
66                .sin_port = htons(PROGGER_SERVER_PORT),
67        };
68
69        progger_release_sock();
70
71        err = sock_create_kern(net, family, SOCK_STREAM, IPPROTO_TCP, &sock);
72        if (err)
73                return err;
74
75        sock_set_reuseaddr(sock->sk);
76
77        if (family == AF_INET6) {
78                err = ip6_sock_set_v6only(sock->sk);
79                if (err)
80                        return err;
81
82                dstaddr = (struct sockaddr *)&dstaddr6;
83                dstaddrlen = sizeof(dstaddr6);
84        } else {
85                dstaddr = (struct sockaddr *)&dstaddr4;
86                dstaddrlen = sizeof(dstaddr4);
87        }
88
89        err = kernel_connect(sock, dstaddr, dstaddrlen, 0);
90        if (err)
91                return err;
92
93        sock->sk->sk_sndbuf = INT_MAX;
94        sock->sk->sk_allocation = GFP_ATOMIC;
95
96        sock_ready = 1;
97
98        return 0;
99 }
100
101 static int conn_loop(void *data)
102 {
103        while (1) {
104                set_current_state(TASK_RUNNING);
105
106                if (kthread_should_stop())
107                        return 0;
108
109                if (kthread_should_park()) {
110                        kthread_parkme();
111                        continue;
112                }
113
114                if (!sock_ready) {
115                        if (progger_connect() == 0)
116                                pr_info("TCP connection established.\n");
117                }
118
119                set_current_state(TASK_INTERRUPTIBLE);
120                schedule_timeout_interruptible(msecs_to_jiffies(100));
121        }
122 }
123
124 int send_encrypted(void *data, size_t len)
```

```
{
	int err;
	void *crypt_output;
	size_t crypt_output_len;
	struct msghdr msg = {};
	struct kvec iov[2];
	struct record_ad ad;
	size_t total_len;

	if (!sock_ready)
		return -EAGAIN;

	ad.len = len;
	ad.len += XCHACHA20POLY1305_NONCE_SIZE + CHACHA20POLY1305_AUTHTAG_SIZE;
	ad.client_id = CONFIG_PROGGER_CLIENT_ID;

	err = progger_xchacha20poly1305(&crypt_output, &crypt_output_len,
					data, len, &ad, sizeof(ad));

	if (unlikely(err))
		return err;

	iov[0].iov_base = &ad;
	iov[0].iov_len = sizeof(ad);
	iov[1].iov_base = crypt_output;
	iov[1].iov_len = crypt_output_len;

	if (unlikely(check_add_overflow(crypt_output_len, sizeof(ad),
					&total_len))) {
		kvfree(crypt_output);
		return -EOVERFLOW;
	}

	err = kernel_sendmsg(sock, &msg, iov, ARRAY_SIZE(iov), total_len);

	kvfree(crypt_output);

	if (unlikely(err < 0)) {
		if (err == -ECONNRESET || err == -EPIPE) {
			pr_info_ratelimited("TCP connection reset.\n");
			sock_ready = 0;
			return -EAGAIN;
		}

		return err;
	}

	return 0;
}

int __init net_init(void)
{
	return progger_create_task(&tcp_worker, conn_loop, "progger-net");
}

void net_exit(void)
{
	progger_destroy_task(&tcp_worker);
	progger_release_sock();
}
```

A.15 drivers/net/progger/kernel/net.h

```
/* SPDX-License-Identifier: GPL-2.0-only */

#ifndef PROGGER_KERNEL_NET_H
#define PROGGER_KERNEL_NET_H

#include <progger/types.h>

int send_encrypted(void *data, size_t len);
int net_init(void);
void net_exit(void);

#endif /* PROGGER_KERNEL_NET_H */
```

A.16 drivers/net/progger/kernel/ringbuf.c

```
// SPDX-License-Identifier: GPL-2.0-only

#include "ringbuf.h"

#include <asm/barrier.h>
#include <linux/cache.h>
#include <linux/cpu.h>
#include <linux/cpumask.h>
#include <linux/err.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/kthread.h>
#include <linux/mm.h>
#include <linux/percpu.h>
#include <linux/printk.h>
#include <linux/sched.h>
#include <linux/slab.h>
#include <linux/spinlock.h>
#include <linux/topology.h>

#include <progger/compiler.h>
#include <progger/types.h>

#include "kthread.h"
#include "net.h"

static struct task_struct *rb_worker;
struct record_ringbuf __percpu __read_mostly *ring;
DEFINE_SPINLOCK(rb_pde_lock);

static int ringbuf_alloc(struct record_ringbuf *rb, int cpu, size_t size)
{
	if (size < MAX_RECORD_SIZE) {
		size = MAX_RECORD_SIZE * 4;
		pr_info("ringbuf: Alloc size increased to %zu.\n", size);
	}

	rb->buf = kvzalloc_node(size, GFP_KERNEL, cpu_to_node(cpu));
```

```
        if (!rb->buf)
                return -ENOMEM;

        rb->size = size;
        rb->data_end = 0;
        rb->consumer = 0;
        rb->producer = 0;

        return 0;
}

static int send_records(int cpu)
{
        struct record_ringbuf *rb = get_ringbuf(cpu);
        size_t consumer, producer, data_end;
        unsigned int retries = 7;
        void *padded;
        size_t len;
        size_t paddedlen;
        int err;

        consumer = rb->consumer;
        spin_lock(&rb_pde_lock);
        producer = smp_load_acquire(&rb->producer);
        data_end = smp_load_acquire(&rb->data_end);
        spin_unlock(&rb_pde_lock);

        if (rb_is_empty(consumer, producer))
                return 0;

        if (producer > consumer)
                len = producer - consumer;
        else if (likely(data_end != 0))
                len = data_end - consumer + 1;
        else
                return 0;

        /*
         * If ‘len‘ is incorrect due to underflow, it should be caught by
         * checking ‘len > rb->size‘. However, ‘consumer + len‘ could result
         * in ‘len‘ overflowing, returning ‘len‘ to a reasonable value.
         * Hence, the two separate checks are done.
         */
        if (unlikely(len > rb->size || consumer + len > rb->size)) {
                pr_err_once("%s: Length miscalculation!\n", __func__);
                return -E2BIG;
        }

        paddedlen = ALIGN(len, PROGGER_RECORD_PADDING_ALIGN);

        padded = kvzalloc(paddedlen, GFP_KERNEL);
        if (!padded)
                return -ENOMEM;

        memcpy(padded, rb->buf + consumer, len);

        do {
                err = send_encrypted(padded, paddedlen);
        } while (err == -EAGAIN && retries--);

```

```
100         kvfree(padded);
101
102         if (err)
103                 return err;
104
105         rb_mark_consumed(rb, len, consumer);
106
107         return 0;
108 }
109
110 static int rb_send(void *data)
111 {
112         int cpu;
113         size_t i;
114
115         while (1) {
116                 set_current_state(TASK_RUNNING);
117
118                 if (kthread_should_stop())
119                         return 0;
120
121                 if (kthread_should_park()) {
122                         kthread_parkme();
123                         continue;
124                 }
125
126                 for_each_possible_cpu(cpu) {
127                         for (i = 0; i < 2; i++) {
128                                 int err = send_records(cpu);
129
130                                 if (err < 0 && err != -EAGAIN)
131                                         pr_warn_once("%s: err %d.\n", __func__,
132                                                      err);
133                         }
134                 }
135
136                 set_current_state(TASK_INTERRUPTIBLE);
137                 schedule_timeout_interruptible(1);
138         }
139
140         return 0;
141 }
142
143 int __init ringbuf_init(void)
144 {
145         int err;
146         int cpu;
147
148         ring = alloc_percpu(typeof(*ring));
149         if (!ring)
150                 return -ENOMEM;
151
152         for_each_possible_cpu(cpu) {
153                 err = ringbuf_alloc(per_cpu_ptr(ring, cpu), cpu,
154                                     1024 * CONFIG_PROGGER_RINGBUF_SIZE);
155                 if (err)
156                         return err;
157         }
158
159         err = progger_create_task(&rb_worker, rb_send, "progger-rb");
```

```
160         if (err)
161                 return err;
162
163         return 0;
164 }
165
166 void ringbuf_exit(void)
167 {
168         int cpu;
169
170         progger_destroy_task(&rb_worker);
171
172         for_each_possible_cpu(cpu)
173                 kvfree(per_cpu_ptr(ring, cpu)->buf);
174
175         free_percpu(ring);
176 }
```

A.17 drivers/net/progger/kernel/ringbuf.h

```
1  /* SPDX-License-Identifier: GPL-2.0-only */
2
3  #ifndef PROGGER_KERNEL_RINGBUF_H
4  #define PROGGER_KERNEL_RINGBUF_H
5
6  #include <asm/barrier.h>
7  #include <linux/percpu.h>
8  #include <linux/spinlock.h>
9
10 #include <progger/record.h>
11 #include <progger/types.h>
12
13 struct record_ringbuf {
14         void *buf;
15         size_t size;
16         size_t data_end;
17         size_t consumer;
18         size_t producer;
19 };
20
21 extern spinlock_t rb_pde_lock;
22 extern struct record_ringbuf *ring;
23
24 static inline struct record_ringbuf *get_ringbuf(int cpu)
25 {
26         return per_cpu_ptr(ring, cpu);
27 }
28
29 static inline size_t rb_nextpos(struct record_ringbuf *rb, size_t pos,
30                                 size_t len)
31 {
32         size_t newpos = pos + len;
33
34         if (unlikely(newpos + MAX_RECORD_SIZE > rb->size))
35                 newpos = 0;
36
37         return newpos;
```

```
}
static_assert(__alignof__(struct record) == 1, "Record alignment != 1.");

static inline bool rb_has_space_left(struct record_ringbuf *rb,
                                     size_t consumer, size_t producer)
{
        if (producer > consumer) {
                size_t nextpos = rb_nextpos(rb, producer, MAX_RECORD_SIZE);

                return nextpos > producer || nextpos < consumer;
        } else if (consumer > producer) {
                return producer + MAX_RECORD_SIZE < consumer;
        } else {
                return true;
        }
}

static inline bool rb_is_empty(size_t consumer, size_t producer)
{
        return producer == consumer;
}

static inline void rb_mark_produced(struct record_ringbuf *rb, size_t len,
                                    size_t consumer, size_t producer)
{
        spin_lock(&rb_pde_lock);

        if (producer >= consumer)
                smp_store_release(&rb->data_end, producer + len - 1);

        smp_store_release(&rb->producer, rb_nextpos(rb, producer, len));

        spin_unlock(&rb_pde_lock);
}

static inline void rb_mark_consumed(struct record_ringbuf *rb, size_t len,
                                    size_t consumer)
{
        smp_store_release(&rb->consumer, rb_nextpos(rb, consumer, len));
}

int ringbuf_init(void);
void ringbuf_exit(void);

#endif /* PROGGER_KERNEL_RINGBUF_H */
```

A.18 drivers/net/progger/kernel/tpm.c

```
// SPDX-License-Identifier: GPL-2.0-only

#include "tpm.h"

#include <crypto/chacha20poly1305.h>
#include <linux/byteorder/generic.h>
#include <linux/err.h>
#include <linux/errno.h>
#include <linux/init.h>
```

```
10 #include <linux/printk.h>
11 #include <linux/random.h>
12 #include <linux/random.h>
13 #include <linux/slab.h>
14 #include <linux/string.h>
15 #include <linux/tpm.h>
16
17 #include <progger/types.h>
18
19 #define TPM_CC_Load                 0x00000157
20 #define TPM_CC_Unseal               0x0000015e
21 #define TPM_CC_FlushContext         0x00000165
22 #define TPM_CC_StartAuthSession     0x00000176
23 #define TPM_CC_PolicyPCR            0x0000017f
24 #define TPM_RH_NULL                 0x40000007
25 #define TPM_RS_PW                   0x40000009
26 #define TPM_ALG_NULL                0x0010
27 #define TPM_ALG_SHA256              0x000b
28 #define TPM_SE_POLICY               0x01
29
30 #define TPM_SESSION_ATTR_RETAIN     (1 << 0)
31
32 /* PC clients are specified to have at least 24 PCRs, and 24 / 8 = 3. */
33 #define PCR_SELECT_MIN              3
34 #define TPM_NONCE_SIZE              0x20
35
36 static struct tpm_chip *tpm_chip;
37 static u8 crypto_key[CHACHA20POLY1305_KEY_SIZE];
38
39 static u8 public_blob[] = {
40 #include "generated/tpm-public-blob.h"
41 };
42
43 static u8 private_blob[] = {
44 #include "generated/tpm-private-blob.h"
45 };
46
47 struct tpm_state {
48         u32 blob_handle;
49         u32 auth_session_handle;
50         u8 auth_session_nonce[TPM_NONCE_SIZE];
51 };
52
53 static int tpm_get_response_item(struct tpm_buf *buf, size_t item_offset,
54                                  void *out, size_t outlen)
55 {
56         u32 response_size = be32_to_cpu(((struct tpm_header
            *)buf->data)->length);
57
58         if ((u64)item_offset + outlen > response_size)
59                 return -E2BIG;
60
61         memcpy(out, buf->data + item_offset, outlen);
62
63         return 0;
64 }
65
66 static int get_nonce(u8 (*nonce)[TPM_NONCE_SIZE])
67 {
68         return get_random_bytes_wait(*nonce, sizeof(*nonce));
```

```
69  }
70
71  static void append_auth_cmd(struct tpm_buf *buf, u32 session_handle,
72                              u8 *nonce, u16 noncelen, u8 session_attrs,
73                              u8 *auth, u16 authlen)
74  {
75          tpm_buf_append_u32(buf, 9 + authlen + noncelen);
76          tpm_buf_append_u32(buf, session_handle);
77
78          if (nonce && noncelen) {
79                  tpm_buf_append_u16(buf, noncelen);
80                  tpm_buf_append(buf, nonce, noncelen);
81          } else {
82                  tpm_buf_append_u16(buf, 0);
83          }
84
85          tpm_buf_append_u8(buf, session_attrs);
86
87          if (auth && authlen) {
88                  tpm_buf_append_u16(buf, authlen);
89                  tpm_buf_append(buf, auth, authlen);
90          } else {
91                  tpm_buf_append_u16(buf, 0);
92          }
93  }
94
95  static int start_auth_session(struct tpm_state *tpm_state)
96  {
97          int err;
98          __be32 handle;
99          struct tpm_buf buf;
100
101         err = get_nonce(&tpm_state->auth_session_nonce);
102         if (err)
103                 return err;
104
105         err = tpm_buf_init(&buf, TPM2_ST_NO_SESSIONS, TPM_CC_StartAuthSession);
106         if (err)
107                 return err;
108
109         tpm_buf_append_u32(&buf, TPM_RH_NULL);
110         tpm_buf_append_u32(&buf, TPM_RH_NULL);
111
112         tpm_buf_append_u16(&buf, sizeof(tpm_state->auth_session_nonce));
113         tpm_buf_append(&buf, tpm_state->auth_session_nonce,
114                        sizeof(tpm_state->auth_session_nonce));
115
116         tpm_buf_append_u16(&buf, 0);
117         tpm_buf_append_u8(&buf, TPM_SE_POLICY);
118         tpm_buf_append_u16(&buf, TPM_ALG_NULL);
119         tpm_buf_append_u16(&buf, TPM_ALG_SHA256);
120
121         err = tpm_send(tpm_chip, buf.data, tpm_buf_length(&buf));
122         if (err)
123                 goto fail;
124
125         err = tpm_get_response_item(&buf, TPM_HEADER_SIZE + 0,
126                                     &handle, sizeof(handle));
127         if (err)
128                 goto fail;
```

```
129
130         tpm_state->auth_session_handle = be32_to_cpu(handle);
131
132 fail:
133         tpm_buf_destroy(&buf);
134         return err;
135 }
136
137 static inline int is_blob_size_congruent(u8 *blob, size_t expected)
138 {
139         if (expected < sizeof(__be16))
140                 return false;
141
142         return be16_to_cpup((__be16 *)blob) == expected - sizeof(__be16);
143 }
144
145 static inline void print_blob_size_err(const char *blob_name)
146 {
147         pr_err("TPM: Length embedded in the blob '%s' doesn't match the length "
148                "of the blob given.\n", blob_name);
149 }
150
151 static int load_blob(struct tpm_state *tpm_state, u32 keyhandle,
152                      u8 *private, size_t private_size,
153                      u8 *public, size_t public_size)
154 {
155         int err;
156         __be32 handle;
157         struct tpm_buf buf;
158
159         if (!is_blob_size_congruent(private, private_size)) {
160                 print_blob_size_err("private");
161                 return -EFAULT;
162         }
163
164         if (!is_blob_size_congruent(public, public_size)) {
165                 print_blob_size_err("public");
166                 return -EFAULT;
167         }
168
169         err = tpm_buf_init(&buf, TPM2_ST_SESSIONS, TPM_CC_Load);
170         if (err)
171                 return err;
172
173         tpm_buf_append_u32(&buf, keyhandle);
174
175         append_auth_cmd(&buf, TPM_RS_PW, NULL, 0, 0, NULL, 0);
176
177         tpm_buf_append(&buf, public, public_size);
178         tpm_buf_append(&buf, private, private_size);
179
180         if (buf.flags & TPM_BUF_OVERFLOW) {
181                 pr_err("TPM: Blob is too large.\n");
182                 err = -E2BIG;
183                 goto fail;
184         }
185
186         err = tpm_send(tpm_chip, buf.data, tpm_buf_length(&buf));
187         if (err)
188                 goto fail;
```

```
189
190         err = tpm_get_response_item(&buf, TPM_HEADER_SIZE + 0,
191                                     &handle, sizeof(handle));
192         if (err)
193                 goto fail;
194
195         tpm_state->blob_handle = be32_to_cpu(handle);
196
197 fail:
198         tpm_buf_destroy(&buf);
199         return err;
200 }
201
202 static int policy_pcr_extend(struct tpm_state *tpm_state)
203 {
204         int err;
205         struct tpm_buf buf;
206         u32 pcr = CONFIG_PROGGER_TPM_PCR;
207         u8 pcr_selection[PCR_SELECT_MIN] = {};
208
209         /* Kconfig should enforce this, but just to be sure... */
210         if (pcr < 8 || pcr > 15) {
211                 pr_err("TPM: Invalid PCR, must be in the range [8, 15].\n");
212                 return -EINVAL;
213         }
214
215         pcr_selection[1] |= 1U << (pcr % 8);
216
217         err = tpm_buf_init(&buf, TPM2_ST_NO_SESSIONS, TPM_CC_PolicyPCR);
218         if (err)
219                 return err;
220
221         tpm_buf_append_u32(&buf, tpm_state->auth_session_handle);
222
223         tpm_buf_append_u16(&buf, 0);
224
225         tpm_buf_append_u32(&buf, 1);
226         tpm_buf_append_u16(&buf, TPM_ALG_SHA256);
227         tpm_buf_append_u8(&buf, sizeof(pcr_selection));
228         tpm_buf_append(&buf, pcr_selection, sizeof(pcr_selection));
229
230         err = tpm_send(tpm_chip, buf.data, tpm_buf_length(&buf));
231
232         tpm_buf_destroy(&buf);
233         return err;
234 }
235
236 static int tpm_get_unsealed_data(struct tpm_buf *buf, void *out,
237                                  size_t expected_size)
238 {
239         int err;
240         u16 size;
241         __be16 response_size;
242
243         err = tpm_get_response_item(buf, TPM_HEADER_SIZE + 4, &response_size,
244                                     sizeof(response_size));
245         if (err)
246                 return err;
247
248         size = be16_to_cpu(response_size);
```

```
249
250         if (size != expected_size) {
251                 pr_err("Unsealed data was %u bytes, expected %zu.\n",
252                        size, expected_size);
253                 return -EINVAL;
254         }
255
256         return tpm_get_response_item(buf, TPM_HEADER_SIZE + 6, out, size);
257 }
258
259 static int tpm_unseal(struct tpm_state *tpm_state, void *out, size_t outlen)
260 {
261         int err;
262         struct tpm_buf buf;
263
264         err = tpm_buf_init(&buf, TPM2_ST_SESSIONS, TPM_CC_Unseal);
265         if (err)
266                 return err;
267
268         tpm_buf_append_u32(&buf, tpm_state->blob_handle);
269         append_auth_cmd(&buf, tpm_state->auth_session_handle,
270                         tpm_state->auth_session_nonce,
271                         sizeof(tpm_state->auth_session_nonce),
272                         TPM_SESSION_ATTR_RETAIN,
273                         NULL, 0);
274
275         err = tpm_send(tpm_chip, buf.data, tpm_buf_length(&buf));
276         if (err)
277                 goto fail;
278
279         err = tpm_get_unsealed_data(&buf, out, outlen);
280         if (err)
281                 goto fail;
282
283 fail:
284         tpm_buf_destroy(&buf);
285         return err;
286 }
287
288 static int flush_context(u32 handle)
289 {
290         int err;
291         struct tpm_buf buf;
292
293         err = tpm_buf_init(&buf, TPM2_ST_NO_SESSIONS, TPM_CC_FlushContext);
294         if (err)
295                 return err;
296
297         tpm_buf_append_u32(&buf, handle);
298         err = tpm_send(tpm_chip, buf.data, tpm_buf_length(&buf));
299
300         tpm_buf_destroy(&buf);
301         return err;
302 }
303
304 static int pcr_extend(u32 pcr)
305 {
306         int i;
307         int res = 0;
308         struct tpm_digest *digests;
```

```
309         u8 hash[TPM_DIGEST_SIZE] = { 0 };
310
311         digests = kcalloc(tpm_chip->nr_allocated_banks, sizeof(*digests),
312                           GFP_KERNEL);
313         if (!digests)
314                 return -ENOMEM;
315
316         for (i = 0; i < tpm_chip->nr_allocated_banks; i++) {
317                 digests[i].alg_id = tpm_chip->allocated_banks[i].alg_id;
318                 memcpy(digests[i].digest, hash, TPM_DIGEST_SIZE);
319         }
320
321         res = tpm_pcr_extend(tpm_chip, pcr, digests);
322
323         kfree(digests);
324         return res;
325 }
326
327 static int tpm_unseal_and_lock_key(void)
328 {
329         int err;
330         struct tpm_state tpm_state = {};
331
332         err = start_auth_session(&tpm_state);
333         if (err)
334                 goto fail;
335
336         pr_debug("TPM: Auth session: 0x%08x.\n", tpm_state.auth_session_handle);
337
338         err = load_blob(&tpm_state, CONFIG_PROGGER_TPM_KEY_HANDLE,
339                         public_blob, sizeof(public_blob),
340                         private_blob, sizeof(private_blob));
341         if (err)
342                 goto fail;
343
344         pr_debug("TPM: Blob: 0x%08x.\n", tpm_state.blob_handle);
345
346         err = policy_pcr_extend(&tpm_state);
347         if (err)
348                 goto fail;
349
350         err = tpm_unseal(&tpm_state, crypto_key, sizeof(crypto_key));
351         if (err)
352                 goto fail;
353
354         err = pcr_extend(CONFIG_PROGGER_TPM_PCR);
355         if (err)
356                 goto fail;
357
358 fail:
359         if (tpm_state.auth_session_handle) {
360                 pr_debug("TPM: Flushing auth session context.\n");
361                 if (flush_context(tpm_state.auth_session_handle) != 0)
362                         pr_warn("Failed to flush context.\n");
363         }
364
365         if (tpm_state.blob_handle) {
366                 pr_debug("TPM: Flushing blob context.\n");
367                 if (flush_context(tpm_state.blob_handle) != 0)
368                         pr_warn("Failed to flush context.\n");
```

```
369         }
370
371         if (err < 0)
372                 return err;
373
374         if (err > 0)
375                 return -ENOTRECOVERABLE;
376
377         return 0;
378 }
379
380 int tpm_get_chacha20poly1305_key(u8 (*key)[CHACHA20POLY1305_KEY_SIZE])
381 {
382         static_assert(sizeof(*key) == sizeof(crypto_key));
383         memcpy(*key, crypto_key, sizeof(*key));
384
385         return 0;
386 }
387
388 int __init tpm_init(void)
389 {
390         int err;
391         int is_tpm2;
392
393         tpm_chip = tpm_default_chip();
394
395         if (IS_ERR_OR_NULL(tpm_chip)) {
396                 if (IS_ENABLED(CONFIG_PROGGER_PANIC_WHEN_NO_TPM_FOUND))
397                         panic("Unable to find TPM!\n");
398
399                 return -ENODEV;
400         }
401
402         is_tpm2 = tpm_is_tpm2(tpm_chip);
403
404         if (is_tpm2 != 1) {
405                 if (IS_ENABLED(CONFIG_PROGGER_PANIC_WHEN_TPM_IS_NOT_VERSION_2))
406                         panic("TPM: tpm_is_tpm2 returned %d.", is_tpm2);
407
408                 if (is_tpm2 < 0)
409                         return is_tpm2;
410
411                 pr_err("TPM 2.0 is required.\n");
412                 return -ENODEV;
413         }
414
415         err = tpm_unseal_and_lock_key();
416
417         if (err) {
418                 if (IS_ENABLED(CONFIG_PROGGER_PANIC_WHEN_KEY_UNSECURED))
419                         panic("TPM: crypto key is unsecured!\n");
420
421                 pr_alert("TPM: crypto key is unsecured!\n");
422                 return err;
423         }
424
425         pr_info("TPM: Key unsealed and locked.\n");
426
427         return 0;
428 }
```

```
429
430 void tpm_exit(void)
431 {
432         if (tpm_chip)
433                 put_device(&tpm_chip->dev);
434
435         memset(crypto_key, 0, sizeof(crypto_key));
436 }
```

A.19 drivers/net/progger/kernel/tpm.h

```
1  /* SPDX-License-Identifier: GPL-2.0-only */
2
3  #ifndef PROGGER_KERNEL_TPM_H
4  #define PROGGER_KERNEL_TPM_H
5
6  #include <crypto/chacha20poly1305.h>
7  #include <linux/errno.h>
8  #include <linux/kconfig.h>
9
10 #include <progger/types.h>
11
12 #if IS_ENABLED(CONFIG_PROGGER_USE_TPM)
13 int tpm_get_chacha20poly1305_key(u8 (*)[CHACHA20POLY1305_KEY_SIZE]);
14 int tpm_init(void);
15 void tpm_exit(void);
16 #else
17 static inline int
18 tpm_get_chacha20poly1305_key(u8 (*key)[CHACHA20POLY1305_KEY_SIZE])
19 {
20         return -EPERM;
21 }
22
23 static inline int tpm_init(void)
24 {
25         return 0;
26 }
27
28 static inline void tpm_exit(void)
29 {
30 }
31 #endif /* IS_ENABLED(CONFIG_PROGGER_USE_TPM) */
32
33 #endif /* PROGGER_KERNEL_TPM_H */
```

A.20 drivers/net/progger/kernel/tracepoints.c

```
1 // SPDX-License-Identifier: GPL-2.0-only
2
3 #include "tracepoints.h"
4
5 #include <asm/barrier.h>
6 #include <asm/current.h>
7 #include <asm/ptrace.h>
```

```
 8 #include <asm/syscall.h>
 9 #include <linux/bits.h>
10 #include <linux/cache.h>
11 #include <linux/cred.h>
12 #include <linux/errno.h>
13 #include <linux/init.h>
14 #include <linux/list.h>
15 #include <linux/pid.h>
16 #include <linux/printk.h>
17 #include <linux/slab.h>
18 #include <linux/smp.h>
19 #include <linux/string.h>
20 #include <linux/timekeeping.h>
21 #include <linux/tracepoint.h>
22 #include <linux/uaccess.h>
23 #include <linux/uidgid.h>
24
25 #include <progger/compiler.h>
26 #include <progger/record.h>
27 #include <progger/types.h>
28
29 #include "ringbuf.h"
30
31 struct progger_syscall_info {
32         /*
33          * Non-zero if the syscall is being traced by Progger.
34          */
35         u8 being_traced;
36
37         /*
38          * If bit n is set, then the nth argument to the syscall is a C string.
39          */
40         u8 cstr_args;
41
42         /*
43          * A bitmask indicating the tracepoints to generate records from.
44          */
45         u8 tp_srcs;
46 };
47
48 /*
49  * If ‘tp_srcs‘ is left unset, it will be later set to the default of
50  * ‘TP_SRC_SYS_EXIT‘.
51  *
52  * See include/linux/syscalls.h and the man page for each syscall.
53  */
54 static struct progger_syscall_info __read_mostly
55 syscalls_info[__NR_syscall_max + 1] = {
56         [__NR_setxattr]        = { .cstr_args = BIT(0) | BIT(1) | BIT(2) },
57         [__NR_lsetxattr]       = { .cstr_args = BIT(0) | BIT(1) | BIT(2) },
58         [__NR_fsetxattr]       = { .cstr_args = BIT(1) | BIT(2) },
59         [__NR_getxattr]        = { .cstr_args = BIT(0) | BIT(1) },
60         [__NR_lgetxattr]       = { .cstr_args = BIT(0) | BIT(1) },
61         [__NR_fgetxattr]       = { .cstr_args = BIT(1) },
62         [__NR_listxattr]       = { .cstr_args = BIT(0) },
63         [__NR_llistxattr]      = { .cstr_args = BIT(0) },
64         [__NR_removexattr]     = { .cstr_args = BIT(0) | BIT(1) },
65         [__NR_lremovexattr]    = { .cstr_args = BIT(0) | BIT(1) },
66         [__NR_fremovexattr]    = { .cstr_args = BIT(1) },
67         [__NR_inotify_add_watch] = { .cstr_args = BIT(1) },
```

```
68          [__NR_mknodat]          = { .cstr_args = BIT(1) },
69          [__NR_mkdirat]          = { .cstr_args = BIT(1) },
70          [__NR_unlinkat]         = { .cstr_args = BIT(1) },
71          [__NR_symlinkat]        = { .cstr_args = BIT(0) | BIT(2) },
72          [__NR_linkat]           = { .cstr_args = BIT(1) | BIT(3) },
73          [__NR_renameat]         = { .cstr_args = BIT(1) | BIT(3) },
74          [__NR_umount2]          = { .cstr_args = BIT(0) },
75          [__NR_mount]            = { .cstr_args = BIT(0) | BIT(1) | BIT(2) },
76          [__NR_pivot_root]       = { .cstr_args = BIT(0) | BIT(1) },
77          [__NR_statfs]           = { .cstr_args = BIT(0) },
78          [__NR_truncate]         = { .cstr_args = BIT(0) },
79          [__NR_faccessat]        = { .cstr_args = BIT(1) },
80          [__NR_faccessat2]       = { .cstr_args = BIT(1) },
81          [__NR_chdir]            = { .cstr_args = BIT(0) },
82          [__NR_chroot]           = { .cstr_args = BIT(0) },
83          [__NR_fchmodat]         = { .cstr_args = BIT(1) },
84          [__NR_fchownat]         = { .cstr_args = BIT(1) },
85          [__NR_openat]           = { .cstr_args = BIT(1) },
86          [__NR_openat2]          = { .cstr_args = BIT(1) },
87          [__NR_quotactl]         = { .cstr_args = BIT(1) },
88          [__NR_readlinkat]       = { .cstr_args = BIT(1) },
89          [__NR_newfstatat]       = { .cstr_args = BIT(1) },
90          [__NR_utimensat]        = { .cstr_args = BIT(1) },
91          [__NR_acct]             = { .cstr_args = BIT(0) },
92          [__NR_init_module]      = { .cstr_args = BIT(2) },
93          [__NR_delete_module]    = { .cstr_args = BIT(0) },
94          [__NR_sethostname]      = { .cstr_args = BIT(0) },
95          [__NR_setdomainname]    = { .cstr_args = BIT(0) },
96          [__NR_mq_open]          = { .cstr_args = BIT(0) },
97          [__NR_mq_unlink]        = { .cstr_args = BIT(0) },
98          [__NR_add_key]          = { .cstr_args = BIT(0) | BIT(1) },
99          [__NR_request_key]      = { .cstr_args = BIT(0) | BIT(1) | BIT(2) },
100         [__NR_execve] = {
101                 .cstr_args = BIT(0),
102                 /*
103                  * ‘execve‘ doesn’t return to the caller when it succeeds,
104                  * and as a result the arguments available in the ‘sys_exit‘
105                  * tracepoint are all zero. To get any useful information,
106                  * the arguments must be copied during ‘sys_enter‘.
107                  *
108                  * ‘TP_SRC_SYS_EXIT‘ is added so that failed ‘execve‘ calls
109                  * can be detected. It does add some noise for successful
110                  * calls, but that noise can be easily filtered out by the
111                  * program that processes the records.
112                  */
113                 .tp_srcs = TP_SRC_SYS_ENTER | TP_SRC_SYS_EXIT,
114         },
115         [__NR_swapon]           = { .cstr_args = BIT(0) },
116         [__NR_swapoff]          = { .cstr_args = BIT(0) },
117         [__NR_fanotify_mark]    = { .cstr_args = BIT(4) },
118         [__NR_name_to_handle_at] = { .cstr_args = BIT(1) },
119         [__NR_finit_module]     = { .cstr_args = BIT(1) },
120         [__NR_renameat2]        = { .cstr_args = BIT(1) | BIT(3) },
121         [__NR_memfd_create]     = { .cstr_args = BIT(0) },
122         [__NR_execveat] = {
123                 .cstr_args = BIT(1),
124                 /*
125                  * See the entry for ‘__NR_execve‘ to see why this is done.
126                  */
127                 .tp_srcs = TP_SRC_SYS_ENTER | TP_SRC_SYS_EXIT,
```

```
128         },
129         [__NR_statx]    = { .cstr_args = BIT(1) },
130         [__NR_open_tree] = { .cstr_args = BIT(1) },
131         [__NR_move_mount] = { .cstr_args = BIT(1) | BIT(3) },
132         [__NR_fsopen]   = { .cstr_args = BIT(0) },
133         [__NR_fsconfig] = { .cstr_args = BIT(2) | BIT(3) },
134         [__NR_fspick]   = { .cstr_args = BIT(1) },
135
136         /* Deprecated syscalls. */
137         [__NR_open]          = { .cstr_args = BIT(0) },
138         [__NR_link]          = { .cstr_args = BIT(0) | BIT(1) },
139         [__NR_unlink]        = { .cstr_args = BIT(0) },
140         [__NR_mknod]         = { .cstr_args = BIT(0) },
141         [__NR_chmod]         = { .cstr_args = BIT(0) },
142         [__NR_chown]         = { .cstr_args = BIT(0) },
143         [__NR_mkdir]         = { .cstr_args = BIT(0) },
144         [__NR_rmdir]         = { .cstr_args = BIT(0) },
145         [__NR_lchown]        = { .cstr_args = BIT(0) },
146         [__NR_access]        = { .cstr_args = BIT(0) },
147         [__NR_rename]        = { .cstr_args = BIT(0) | BIT(1) },
148         [__NR_symlink]       = { .cstr_args = BIT(0) | BIT(1) },
149         [__NR_utime]         = { .cstr_args = BIT(0) },
150         [__NR_utimes]        = { .cstr_args = BIT(0) },
151         [__NR_futimesat]     = { .cstr_args = BIT(1) },
152         [__NR_creat]         = { .cstr_args = BIT(0) },
153         [__NR_uselib]        = { .cstr_args = BIT(0) },
154         [__NR_kexec_file_load] = { .cstr_args = BIT(3) },
155         [__NR_stat]          = { .cstr_args = BIT(0) },
156         [__NR_lstat]         = { .cstr_args = BIT(0) },
157         [__NR_readlink]      = { .cstr_args = BIT(0) },
158 };
159
160 static inline bool arg_is_cstr(const struct progger_syscall_info info,
161                                u8 argnum)
162 {
163         return info.cstr_args & BIT(argnum);
164 }
165
166 static inline void record_add_str_user(struct record *record,
167                                        const char __user *str)
168 {
169         char *dst;
170         ssize_t len;
171
172         if (!str)
173                 return;
174
175         if (unlikely(record->len >= MAX_RECORD_SIZE))
176                 return;
177
178         dst = (char *)record + record->len;
179
180         pagefault_disable();
181         len = strncpy_from_user(dst, str, MAX_RECORD_STR_SIZE);
182         pagefault_enable();
183
184         if (unlikely(len < 0))
185                 return;
186
187         if (unlikely(len == MAX_RECORD_STR_SIZE)) {
```

```
188                 dst[MAX_RECORD_STR_SIZE - 1] = '\0';
189                 record->len += len;
190         } else {
191                 record->len += len + 1;
192         }
193 }
194
195 static inline unsigned long get_arg_x86_64(struct pt_regs *regs,
196                                            unsigned int n)
197 {
198         static const unsigned int argument_offs[] = {
199                 offsetof(struct pt_regs, di),
200                 offsetof(struct pt_regs, si),
201                 offsetof(struct pt_regs, dx),
202                 offsetof(struct pt_regs, r10),
203                 offsetof(struct pt_regs, r8),
204                 offsetof(struct pt_regs, r9),
205         };
206
207         return regs_get_register(regs, argument_offs[n]);
208 }
209
210 static void syscall_tp(struct pt_regs *regs, u8 tp_src)
211 {
212         int cpu;
213         long id;
214         size_t i;
215         struct progger_syscall_info syscall;
216         struct record *record;
217         struct record_syscall_x86_64 *data;
218         struct record_ringbuf *rb;
219         size_t producer, consumer;
220
221         id = syscall_get_nr(current, regs);
222
223         if (id < 0 || id >= ARRAY_SIZE(syscalls_info))
224                 return;
225
226         syscall = syscalls_info[id];
227
228         if (!syscall.being_traced)
229                 return;
230
231         if (!(syscall.tp_srcs & tp_src))
232                 return;
233
234         cpu = smp_processor_id();
235         rb = get_ringbuf(cpu);
236
237         producer = rb->producer;
238         consumer = smp_load_acquire(&rb->consumer);
239
240         if (unlikely(!rb_has_space_left(rb, consumer, producer))) {
241                 pr_warn_once("%s: Ringbuf overflow.\n", __func__);
242                 return;
243         }
244
245         record = (struct record *)(rb->buf + producer);
246
247         record->len = sizeof(*record) + sizeof(*data);
```

```
248         record->id = RECORD_SYSCALL_X86_64;
249
250         data = (struct record_syscall_x86_64 *)record->data;
251
252         data->nr = id;
253         data->tp_src = tp_src;
254         data->ts = ktime_get_mono_fast_ns();
255         data->ret = tp_src == TP_SRC_SYS_EXIT ? regs_return_value(regs) : 0;
256
257         data->pid = current->pid;
258         data->uid = __kuid_val(current_uid());
259         data->euid = __kuid_val(current_euid());
260
261         for (i = 0; i < ARRAY_SIZE(data->args); i++) {
262                 unsigned long arg = get_arg_x86_64(regs, i);
263
264                 data->args[i] = arg;
265
266                 if (arg_is_cstr(syscall, i))
267                         record_add_str_user(record, (const char __user *)arg);
268         }
269
270         rb_mark_produced(rb, record->len, consumer, producer);
271 }
272
273 static void __used sys_enter_tp(void *p, struct pt_regs *regs, long id)
274 {
275         syscall_tp(regs, TP_SRC_SYS_ENTER);
276 }
277
278 static void __used sys_exit_tp(void *p, struct pt_regs *regs, long ret)
279 {
280         syscall_tp(regs, TP_SRC_SYS_EXIT);
281 }
282
283 #include "generated/syscalls.h"
284
285 struct tp_item {
286         struct list_head list;
287         struct tracepoint *tp;
288         void *probe;
289         void *data;
290         const char *name;
291 };
292
293 static LIST_HEAD(tp_list);
294
295 static int set_traced_syscalls(void)
296 {
297         size_t i;
298
299         for (i = 0; i < ARRAY_SIZE(syscalls_info); i++)
300                 syscalls_info[i].being_traced = 0;
301
302         for (i = 0; i < ARRAY_SIZE(init_syscalls); i++) {
303                 long nr = init_syscalls[i];
304
305                 if (nr < 0 || nr >= ARRAY_SIZE(syscalls_info))
306                         return -EINVAL;
307
```

```
308                 syscalls_info[nr].being_traced = 1;
309         }
310
311         return 0;
312 }
313
314 static inline int tp_item_register(struct tp_item *tp)
315 {
316         if (!tp->tp)
317                 return -EFAULT;
318
319         return tracepoint_probe_register(tp->tp, tp->probe, tp->data);
320 }
321
322 static inline int tp_item_unregister(struct tp_item *tp)
323 {
324         if (!tp->tp)
325                 return -EFAULT;
326
327         return tracepoint_probe_unregister(tp->tp, tp->probe, tp->data);
328 }
329
330 static void tp_search_fn(struct tracepoint *tp, void *priv)
331 {
332         struct tp_item *tp_item = (struct tp_item *)priv;
333
334         if (strcmp(tp->name, tp_item->name) == 0)
335                 tp_item->tp = tp;
336 }
337
338 static int tp_register(void *probe, const char *name)
339 {
340         int err;
341         struct tp_item *new = kzalloc(sizeof(*new), GFP_KERNEL);
342
343         if (!new)
344                 return -ENOMEM;
345
346         new->probe = probe;
347         new->data = NULL;
348         new->name = name;
349
350         for_each_kernel_tracepoint(tp_search_fn, new);
351
352         if (!new->tp) {
353                 pr_err("Couldn't locate tracepoint: %s.\n", name);
354                 kfree(new);
355                 return -EINVAL;
356         }
357
358         err = tp_item_register(new);
359         if (err) {
360                 kfree(new);
361                 return err;
362         }
363
364         list_add(&new->list, &tp_list);
365         pr_info("Registered tracepoint: %s\n", new->name);
366
367         return 0;
```

```
}

static void tp_unregister_all(void)
{
	struct tp_item *tp_item;

	list_for_each_entry(tp_item, &tp_list, list) {
		if (tp_item_unregister(tp_item) != 0) {
			pr_warn("Failed to unregister tracepoint: %s\n",
				tp_item->name);
		} else {
			pr_info("Unregistered tracepoint: %s\n", tp_item->name);
		}
	}

	tracepoint_synchronize_unregister();
}

static bool is_sys_enter_tp_needed(void)
{
	size_t i;

	for (i = 0; i < ARRAY_SIZE(init_syscalls); i++) {
		long nr = init_syscalls[i];

		if (syscalls_info[nr].tp_srcs & TP_SRC_SYS_ENTER)
			return true;
	}

	return false;
}

static void set_default_syscall_tp_src(u8 default_tp_src)
{
	size_t i;

	for (i = 0; i < ARRAY_SIZE(syscalls_info); i++) {
		if (!syscalls_info[i].tp_srcs)
			syscalls_info[i].tp_srcs = default_tp_src;
	}
}

int __init tracepoint_init(void)
{
	int err;
	size_t i;

	/*
	 * Set the list of traced syscalls before enbling the tracepoints,
	 * so that we don't end up momentarily generating data for only
	 * subsets of the syscalls.
	 */
	err = set_traced_syscalls();
	if (err)
		return err;

	set_default_syscall_tp_src(TP_SRC_SYS_EXIT);

	for (i = 0; i < ARRAY_SIZE(init_tracepoints); i++) {
		void *fn = init_tracepoints[i].fn;
```

```
428                 const char *name = init_tracepoints[i].name;
429
430                 if (fn == sys_enter_tp && !is_sys_enter_tp_needed())
431                         continue;
432
433                 err = tp_register(fn, name);
434                 if (err) {
435                         pr_err("Failed to register tracepoint: %s.\n", name);
436                         return err;
437                 }
438
439                 pr_debug("Registered tracepoint: %s.\n", name);
440         }
441
442         return 0;
443 }
444
445 void tracepoint_exit(void)
446 {
447         tp_unregister_all();
448 }
```

A.21 drivers/net/progger/kernel/tracepoints.h

```
1 /* SPDX-License-Identifier: GPL-2.0-only */
2
3 #ifndef PROGGER_KERNEL_TRACEPOINTS_H
4 #define PROGGER_KERNEL_TRACEPOINTS_H
5
6 int tracepoint_init(void);
7 void tracepoint_exit(void);
8
9 #endif /* PROGGER_KERNEL_TRACEPOINTS_H */
```

A.22 drivers/net/progger/scripts/tpm/provision

```
1  #!/usr/bin/env bash
2  # SPDX-License-Identifier: GPL-2.0-only
3
4  script_dir="$(dirname "$(realpath "$0")")"
5
6  set -u
7
8  tmp_dir="$(mktemp -d)"
9  output_path="${1:-sealed-data}"
10 output_dir="$(dirname "${output_path}")"
11 output_basename="$(basename "${output_path}")"
12
13 cmd=(
14         "${script_dir}/provision-setup" "${tmp_dir}" "${output_dir}" '&&'
15         chroot "${tmp_dir}" "${script_dir}/provision-inner" "${output_basename}"
16 )
17
18 unshare --mount bash -c 'eval "$@"' progger-tpm "${cmd[@]}"
```

```
rmdir "${tmp_dir}"
```

A.23 drivers/net/progger/scripts/tpm/provision-inner

```
#!/usr/bin/env bash
# SPDX-License-Identifier: GPL-2.0-only

##
## Do not run this directly unless you're sure of what you're doing.
## Instead, run ./provision, which wraps this script to prevent data leaks.
##
## Limitations:
## - Only accepts password auth for lockdown/owner.
## - The PCR value used in the policy is fixed to whatever it is when this
##   script is run.
## - Assumes the hashalg is SHA256.
##

set -eu
set -o pipefail

keyfile='/tmp/key'

check_installed() {
	local cmd="$1"
	local pkg="$2"

	if ! type "${cmd}" >/dev/null 2>&1; then
		printf >&2 'Cannot find `%s`. Try running `apt install %s`.\n' \
			"${cmd}" "${pkg}"
		return 1
	fi
}

check_tpm2_tools_version() {
	local tmp
	local version
	local minver="$1"

	version="$(tpm2_create --version | grep -Po 'version=".+?"' | \
			cut -d \" -f 2)"
	tmp="$(printf '%s\n' "${version}" "${minver}" | sort -V | head -1)"

	# This is actually checking if ${version} >= ${minver}.
	[[ "${tmp}" == "${minver}" ]]
}

check_swap_space() {
	local swaptotal

	swaptotal="$(grep SwapTotal /proc/meminfo | awk '{print $2}')"

	if [[ "${swaptotal}" != "0" ]]; then
		printf >&2 'Found %d KiB of swap space. ' "${swaptotal}"
```

```
51                printf >&2 'Having swap enabled can lead to memory leaks.\n'
52                printf >&2 'To prevent memory leaks, consider disabling swap '
53                printf >&2 'or using encrypted swap.\n'
54
55                read -r -p 'Continue with swap enabled? [y/N] '
56
57                if ! [[ "${REPLY}" =~ [Yy]([Ee][Ss])? ]]; then
58                        return 1
59                fi
60        fi
61  }
62
63  get_pass() {
64        local prompt="$1"
65        local outvar="$2"
66
67        read -r -s -p "${prompt}" "${outvar}"
68        echo
69
70        if [[ -z "${!outvar}" ]]; then
71                echo >&2 'Value cannot be empty.'
72                return 1
73        fi
74  }
75
76  get_pass_confirm() {
77        local prompt="$1"
78        local outvar="$2"
79
80        get_pass "${prompt}" "${outvar}"
81
82        read -r -s -p 'Repeat to confirm: ' pass_confirmation
83        echo
84
85        if [[ "${!outvar}" != "${pass_confirmation}" ]]; then
86                echo >&2 'Values did not match.'
87                return 1
88        fi
89  }
90
91  auth() {
92        local hierarchy="$1"
93        local authval_var="${1}_pass"
94        shift
95
96        # Ensure that the auth value isn't visible to other processes
97        # though the command line arguments.
98        "$@" -C "${hierarchy}" -P file:- <<<"${!authval_var}"
99  }
100
101 setauth() {
102       local hierarchy="$1"
103       local authval_var="${1}_pass"
104       local authval_file='/tmp/authval'
105       local currauthval_var="${2:-}_pass"
106       local currauthval_file='/tmp/currauthval'
107       local _pass=''
108
109       cat <<<"${!authval_var}" | tr -d '\n' >"${authval_file}"
110       cat <<<"${!currauthval_var}" | tr -d '\n' >"${currauthval_file}"
```

```
111
112         tpm2_changeauth -c "${hierarchy}" \
113                 -p file:"${currauthval_file}" \
114                 file:"${authval_file}"
115
116         rm "${authval_file}"
117         rm "${currauthval_file}"
118 }
119
120 is_auth_set() {
121         local hierarchy="$1"
122
123         ! tpm2_changeauth -c "${hierarchy}" >/dev/null 2>&1
124 }
125
126 get_user_input() {
127         local progger_crypto_key_b64
128         local key_required_len='32'
129         local key_actual_len
130
131         # We only need to ensure that the lockdown auth is set. If it is,
132         # we have no need to know the passphrase.
133         if ! is_auth_set lockout; then
134                 echo 'Lockout auth is unset. Please set it.'
135                 get_pass_confirm 'Lockout passphrase: ' lockout_pass
136                 setauth lockout
137         fi
138
139         if ! is_auth_set 'owner'; then
140                 echo 'Owner auth is unset. Please set it.'
141                 get_pass_confirm 'Owner passphrase: ' owner_pass
142                 setauth owner
143         else
144                 get_pass 'Owner passphrase: ' owner_pass
145
146                 # Set the passphrase to itself to test whether we have
147                 # the correct passphrase.
148                 if ! setauth owner owner; then
149                         echo >&2 'Owner passphrase is incorrect.'
150                         return 1
151                 fi
152         fi
153
154         get_pass_confirm 'Crypto key (base64): ' progger_crypto_key_b64
155
156         read -r -p 'PCR: ' pcr
157         echo
158
159         if (( pcr < 8 )) || (( pcr > 15 )); then
160                 echo >&2 'PCR value must be in the range [8, 15].'
161                 return 1
162         fi
163
164         # Remember that the keyfile is being saved to a location on a tmpfs
165         # that is only visible in this process's mount namespace.
166         base64 -d <<<"${progger_crypto_key_b64}" >"${keyfile}"
167
168         key_actual_len="$(<"${keyfile}" wc -c)"
169         if [[ "${key_required_len}" != "${key_actual_len}" ]]; then
170                 printf >&2 'Crypto key must be %d bytes. Got %d bytes.' \
```

```
                        "${key_required_len}" "${key_actual_len}"
                return 1
        fi
}

provision() {
        local output_stem="$1"
        local hashalg='sha256'
        local persistent_handle
        local tmphash

        tpm2_createpolicy -l "${hashalg}:${pcr}" --policy-pcr \
                --policy /tmp/policy >/dev/null
        auth owner tpm2_createprimary -c /tmp/ctx >/dev/null

        # XXX: The output is YAML, but grep/awk are used.
        persistent_handle="$(auth owner tpm2_evictcontrol -c /tmp/ctx | \
                        grep '^persistent-handle: ' | awk '{print $2}')"

        tpm2_create -C "${persistent_handle}" \
                --public "${output_stem}-public" \
                --private "${output_stem}-private" \
                -g sha256 -L /tmp/policy -i - \
                <"${keyfile}" >/dev/null

        printf 'Persistent handle: %s\n' "${persistent_handle}"

        tpm2_load -C "${persistent_handle}" \
                --public "${output_stem}-public" \
                --private "${output_stem}-private" \
                -c /tmp/load-ctx >/dev/null

        tpm2_unseal -c /tmp/load-ctx -p "pcr:${hashalg}:${pcr}" >/tmp/unseal

        if ! diff "${keyfile}" /tmp/unseal >/dev/null 2>&1; then
                echo >&2 "Unsealed data didn't match sealed data!"
                return 1
        fi

        echo 'Unsealed data matches sealed data.'

        printf 'Extending PCR %d.\n' "${pcr}"
        # It doesn't matter what it's extended to, only that it can't get back.
        tmphash="$(echo -n progger | sha256sum | awk '{print $1}')"
        tpm2_pcrextend "${pcr}:${hashalg}=${tmphash}"
}

main() {
        local output_basename="$1"

        check_installed "tpm2_create" "tpm2-tools" || exit 1
        check_tpm2_tools_version '4.2' || \
                echo >&2 'Warning: tpm2-tools version < 4.2, commands may fail.'

        check_swap_space || exit 1

        get_user_input
        provision "/output/${output_basename}"
}

```

```
main "$@"
```

A.24 drivers/net/progger/scripts/tpm/provision-setup

```
#!/usr/bin/env bash
# SPDX-License-Identifier: GPL-2.0-only

set -eu
set -o pipefail

tmpdir_mount() {
	local dst="${!#}"
	local realdst="${tmpdir}/${dst}"
	local end=$(( $# - 1 ))

	mkdir -p "${realdst}"
	mount "${@:1:$end}" "${realdst}"
}

main() {
	tmpdir="$1"
	local outputdir="$2"

	mount -t tmpfs none "${tmpdir}"

	tmpdir_mount -o bind,rw "${outputdir}" /output
	tmpdir_mount -t tmpfs none /tmp
	tmpdir_mount -t proc proc /proc
	tmpdir_mount -t sysfs sys /sys
	tmpdir_mount --rbind /dev /dev

	for dir in /bin /etc /home /lib /lib32 /lib64 /opt /root \
			/run /sbin /usr /var; do
		if ! [[ -d "${dir}" ]]; then
			continue
		fi

		tmpdir_mount -o bind,ro "${dir}" "${dir}"
	done
}

main "$@"
```

A.25 drivers/net/progger/server/Makefile

```
# SPDX-License-Identifier: GPL-2.0-only

server-cflags += -O2 -std=c11 -march=native -g -fPIE -D_FORTIFY_SOURCE=2
server-cflags += -pipe -fstack-protector-strong -fno-strict-aliasing
server-cflags += -D_DEFAULT_SOURCE -D_GNU_SOURCE
server-cflags += -I $(srctree)/include/progger-host
```

```

server-cflags += -Werror -Wall -Wextra -Wstack-protector -Wformat=2 -Wshadow
server-cflags += -Wundef -Wcast-qual -Wcast-align -Wlogical-op -Winit-self
server-cflags += -Wstrict-overflow=5 -Wredundant-decls -Wnull-dereference
server-cflags += -Wshift-overflow=2 -Wduplicated-cond -Wjump-misses-init
server-cflags += -Wstrict-prototypes -Wwrite-strings

server-cflags += $(shell pkg-config --cflags libsodium)
server-cflags += $(shell pkg-config --cflags json-c)

HOSTLDLIBS_server += $(shell pkg-config --libs libsodium)
HOSTLDLIBS_server += $(shell pkg-config --libs json-c)

HOSTLDLIBS_server += -Wl,-z,relro,-z,now -pie

hostprogs += server

server-objs += server.o
server-objs += crypto.o
server-objs += syscall-table.o

always-y += $(hostprogs)

$(foreach obj,$(server-objs),$(eval HOSTCFLAGS_$(obj) := $(server-cflags)))
```

A.26 drivers/net/progger/server/crypto.c

```
// SPDX-License-Identifier: GPL-2.0-only

#include "crypto.h"

#include <errno.h>
#include <stddef.h>
#include <stdio.h>

#include <sodium.h>

#include <progger/compiler.h>
#include <progger/crypto.h>
#include <progger/record.h>
#include <progger/types.h>

#define NONCE_SIZE    24
#define KEY_SIZE      32
#define AUTHTAG_SIZE  16

struct record *decrypt(void *data, size_t datalen, void *ad, size_t adlen,
                       const unsigned char *key)
{
        int err;
        uint8_t *nonce;
        struct record *record;
        unsigned long long record_len;

        nonce = (uint8_t *)data;

        /* TODO: Could be less if we subtract the nonce and auth tag. */
```

```
31         record = malloc(datalen);
32
33         if (!record) {
34                 perror("malloc");
35                 return NULL;
36         }
37
38         err = crypto_aead_xchacha20poly1305_ietf_decrypt(
39                 (void *)record, &record_len, NULL,
40                 data + NONCE_SIZE,
41                 datalen - NONCE_SIZE,
42                 ad, adlen,
43                 nonce, key);
44
45         if (err) {
46                 fprintf(stderr, "Decryption failed!\n");
47                 free(record);
48                 return NULL;
49         }
50
51         return record;
52 }
```

A.27 drivers/net/progger/server/crypto.h

```
1  /* SPDX-License-Identifier: GPL-2.0-only */
2
3  #ifndef PROGGER_SERVER_CRYPTO_H
4  #define PROGGER_SERVER_CRYPTO_H
5
6  #include <stddef.h>
7
8  struct record *decrypt(void *data, size_t datalen, void *ad, size_t adlen,
9                         const unsigned char *key);
10
11 #endif /* PROGGER_SERVER_CRYPTO_H */
```

A.28 drivers/net/progger/server/server.c

```
1  // SPDX-License-Identifier: GPL-2.0-only
2
3  #include <arpa/inet.h>
4  #include <errno.h>
5  #include <fcntl.h>
6  #include <netdb.h>
7  #include <netinet/in.h>
8  #include <poll.h>
9  #include <signal.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <sys/stat.h>
14 #include <sys/socket.h>
15 #include <sys/types.h>
```

```
16 #include <unistd.h>
17
18 #include <progger/crypto.h>
19 #include <progger/compiler.h>
20 #include <progger/net.h>
21 #include <progger/record.h>
22
23 #include <json.h>
24
25 #include "crypto.h"
26 #include "syscalls.h"
27
28 #define str(s) #s
29 #define xstr_check_arg_defined(s) ({ \
30        (void)s;                      \
31        str(s);                       \
32 })
33 #define xstr(s) xstr_check_arg_defined(s)
34
35 static unsigned char crypto_key[CHACHA20POLY1305_KEY_SIZE];
36
37 /*
38  * TODO: Close the listenfds.
39  */
40 static void exit_cleanly(__unused int n)
41 {
42        putchar('\n');
43        exit(0);
44 }
45
46 static void init_signal_handlers(void)
47 {
48        struct sigaction action_exit_cleanly = {
49               .sa_handler = exit_cleanly,
50        };
51
52        if (sigaction(SIGINT, &action_exit_cleanly, NULL) < 0) {
53               perror("sigation");
54               exit(1);
55        }
56 }
57
58 static const char *addrinfo_ip(const struct addrinfo *addr)
59 {
60        const void *src;
61        static char buf[INET6_ADDRSTRLEN];
62
63        strcpy(buf, "???");
64
65        if (addr->ai_family == AF_INET)
66               src = &((struct sockaddr_in *)addr->ai_addr)->sin_addr;
67        else if (addr->ai_family == AF_INET6)
68               src = &((struct sockaddr_in6 *)addr->ai_addr)->sin6_addr;
69        else
70               return buf;
71
72        if (!inet_ntop(addr->ai_family, src, buf, sizeof(buf)))
73               strcpy(buf, "???");
74
75        return buf;
```

```
}

static int bind_and_listen(struct addrinfo *addr, int backlog)
{
	int sock;
	static const int yes = 1;

	sock = socket(addr->ai_family, addr->ai_socktype, addr->ai_protocol);

	if (sock < 0)
		return -1;

	if (setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &yes, sizeof(yes)) != 0)
		return -1;

	if (bind(sock, addr->ai_addr, addr->ai_addrlen) != 0)
		return -1;

	if (listen(sock, backlog) != 0)
		return -1;

	return sock;
}

/*
 * TODO: json_object_*_add can fail, returning != 0.
 */
static void print_as_json(struct record_syscall_x86_64 *data, size_t data_len)
{
	struct json_object *parent;
	struct json_object *args;
	struct json_object *strs;
	size_t remaining_str_len = data_len - sizeof(*data);
	size_t travelled = 0;
	const char *tp_src_str;

	parent = json_object_new_object();

	json_object_object_add(parent, "id",
		json_object_new_string(syscall_str_from_nr(data->nr)));

	if (data->tp_src == TP_SRC_SYS_ENTER)
		tp_src_str = "sys_enter";
	else if (data->tp_src == TP_SRC_SYS_EXIT)
		tp_src_str = "sys_exit";
	else
		tp_src_str = "unknown";

	json_object_object_add(parent, "tp_src",
		json_object_new_string(tp_src_str));

	json_object_object_add(parent, "ts", json_object_new_int64(data->ts));
	json_object_object_add(parent, "ret", json_object_new_int64(data->ret));

	json_object_object_add(parent, "pid", json_object_new_int(data->pid));
	json_object_object_add(parent, "uid", json_object_new_int(data->uid));
	json_object_object_add(parent, "euid", json_object_new_int(data->euid));

	args = json_object_new_array();

```

```
136         for (size_t i = 0; i < ARRAY_SIZE(data->args); i++)
137                 json_object_array_add(args,
                        json_object_new_int64(data->args[i]));
138
139         json_object_object_add(parent, "args", args);
140
141         strs = json_object_new_array();
142
143         if (remaining_str_len)
144                 data->strings[remaining_str_len - 1] = '\0';
145
146         while (remaining_str_len != 0) {
147                 struct json_object *str;
148
149                 str = json_object_new_string(&data->strings[travelled]);
150                 json_object_array_add(strs, str);
151
152                 travelled += json_object_get_string_len(str) + 1;
153                 remaining_str_len = data_len - sizeof(*data) - travelled;
154
155                 if (travelled > data_len - sizeof(*data)) {
156                         fprintf(stderr, "String len has been miscalculated.\n");
157                         break;
158                 }
159         }
160
161         json_object_object_add(parent, "strings", strs);
162
163         puts(json_object_to_json_string(parent));
164         fflush(stdout);
165
166         json_object_put(parent);
167 }
168
169 static ssize_t do_recv(int sock)
170 {
171         ssize_t ret;
172         ssize_t recvlen;
173         size_t dataleft;
174         static struct record_ad ad;
175         struct record *decrypted;
176         struct record *r;
177         uint8_t *buf;
178
179         recvlen = recv(sock, &ad, sizeof(ad), MSG_PEEK | MSG_DONTWAIT);
180
181         if (recvlen < 0)
182                 return -errno;
183
184         if (recvlen != sizeof(ad))
185                 return 0;
186
187         recvlen = recv(sock, &ad, sizeof(ad), MSG_WAITALL);
188
189         if (recvlen < 0)
190                 return -errno;
191
192         if (recvlen != sizeof(ad))
193                 return -EBADMSG;
194
```

```
195         buf = malloc(ad.len);
196 
197         if (!buf)
198                 return -ENOMEM;
199 
200         recvlen = recv(sock, buf, ad.len, MSG_WAITALL);
201 
202         if (recvlen != ad.len)
203                 return -EBADMSG;
204 
205         if (recvlen < 0)
206                 return -errno;
207 
208         decrypted = decrypt(buf, recvlen, &ad, sizeof(ad), crypto_key);
209 
210         if (!decrypted) {
211                 free(buf);
212                 return -EBADMSG;
213         }
214 
215         dataleft = recvlen;
216         dataleft -= XCHACHA20POLY1305_NONCE_SIZE;
217         dataleft -= CHACHA20POLY1305_AUTHTAG_SIZE;
218 
219         ret = recvlen;
220 
221         for (r = decrypted; r && dataleft >= sizeof(*r); r = next_record(r)) {
222                 if (r->len > dataleft) {
223                         ret = -EBADMSG;
224                         break;
225                 }
226 
227                 dataleft -= r->len;
228 
229                 print_as_json((struct record_syscall_x86_64 *)
230                               r->data, r->len - sizeof(*r));
231         }
232 
233         free(decrypted);
234         free(buf);
235 
236         return ret;
237 }
238 
239 struct connvector {
240         struct pollfd *fds;
241         size_t nmemb;
242         size_t ncons;
243 };
244 
245 #define CONNVECTOR_MAX_NMEMB (1024 * 1024)
246 #define CONNVECTOR_MIN_NMEMB 8
247 
248 static int add_conn(struct connvector *cvec, int fd)
249 {
250         if (cvec->ncons == cvec->nmemb) {
251                 size_t nmemb;
252 
253                 if (cvec->nmemb >= CONNVECTOR_MAX_NMEMB / 2)
254                         return -1;
```

```
255
256                 nmemb = cvec->nmemb ? cvec->nmemb * 2 : CONNVECTOR_MIN_NMEMB;
257                 cvec->fds = reallocarray(cvec->fds, nmemb, sizeof(*cvec->fds));
258
259                 if (!cvec->fds) {
260                         cvec->ncons = 0;
261                         cvec->nmemb = 0;
262                         return -1;
263                 }
264
265                 cvec->nmemb = nmemb;
266         }
267
268         if (!cvec->fds)
269                 return -1;
270
271         cvec->fds[cvec->ncons].fd = fd;
272         cvec->fds[cvec->ncons].events = POLLIN | POLLHUP;
273
274         cvec->ncons++;
275
276         return 0;
277 }
278
279 static int remove_conn(struct connvector *cvec, size_t offset)
280 {
281         if (!cvec->fds)
282                 return -1;
283
284         if (offset >= cvec->ncons || cvec->ncons == 0)
285                 return -1;
286
287         if (offset == cvec->ncons)
288                 cvec->fds[offset] = (struct pollfd){};
289         else
290                 cvec->fds[offset] = cvec->fds[cvec->ncons - 1];
291
292         cvec->ncons--;
293
294         return 0;
295 }
296
297 static void freeconnvector(struct connvector *cvec)
298 {
299         free(cvec->fds);
300 }
301
302 static int listen_on_addr(const char *node)
303 {
304         int err;
305         struct addrinfo *addr;
306         struct addrinfo *addrs = NULL;
307         struct pollfd *listenfds = NULL;
308         struct connvector cvec = {};
309         const int listen_backlog = 128;
310         size_t n_addrs = 0;
311         size_t i = 0;
312         int ret = -1;
313
314         static const struct addrinfo hints = {
```

```
315                 .ai_flags = AI_PASSIVE,
316                 .ai_family = AF_UNSPEC,
317                 .ai_socktype = SOCK_STREAM,
318                 .ai_protocol = IPPROTO_TCP,
319         };
320
321         err = getaddrinfo(node, xstr(PROGGER_SERVER_PORT), &hints, &addrs);
322         if (err) {
323                 fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(err));
324                 goto fail;
325         }
326
327         fprintf(stderr, "Host '%s' resolved to: ", node);
328         for (addr = addrs; addr; addr = addr->ai_next) {
329                 fprintf(stderr, "%s", addrinfo_ip(addr));
330                 if (addr->ai_next)
331                         fprintf(stderr, ", ");
332                 n_addrs++;
333         }
334         fprintf(stderr, ".\n");
335
336         listenfds = calloc(n_addrs, sizeof(*listenfds));
337
338         if (!listenfds) {
339                 perror("calloc");
340                 goto fail;
341         }
342
343         for (addr = addrs; addr; addr = addr->ai_next, i++) {
344                 int fd = bind_and_listen(addr, listen_backlog);
345
346                 if (fd < 0)
347                         goto fail;
348
349                 if (fcntl(fd, F_SETFL, O_NONBLOCK) != 0)
350                         goto fail;
351
352                 listenfds[i].fd = fd;
353                 listenfds[i].events = POLLIN;
354         }
355
356         while (1) {
357                 int nev = poll(listenfds, n_addrs, 0);
358
359                 if (nev < 0) {
360                         perror("poll");
361                         goto fail;
362                 }
363
364                 for (i = 0; nev > 0 && i < n_addrs; i++, nev--) {
365                         int sock;
366
367                         if (!(listenfds[i].revents & POLLIN))
368                                 continue;
369
370                         sock = accept(listenfds[i].fd, NULL, NULL);
371
372                         /* TODO: Take an action if add_conn fails. */
373                         if (sock > 0)
374                                 add_conn(&cvec, sock);
```

```
375                 }
376
377                 nev = poll(cvec.fds, cvec.ncons, 0);
378
379                 if (nev < 0) {
380                         perror("poll");
381                         goto fail;
382                 }
383
384                 /*
385                  * TODO: Connections currently aren't being removed.
386                  */
387                 for (i = 0; nev > 0 && i < cvec.ncons; i++, nev--) {
388                         /* TODO: Take an action if remove_conn fails. */
389                         if (cvec.fds[i].revents & POLLHUP)
390                                 remove_conn(&cvec, i);
391
392                         if (!(cvec.fds[i].revents & POLLIN))
393                                 continue;
394
395                         err = do_recv(cvec.fds[i].fd);
396
397                         /* TODO: Handle more errors. */
398                         switch (err) {
399                         case 0:
400                                 break;
401                         case -EAGAIN:
402 #if EAGAIN != EWOULDBLOCK
403                         case -EWOULDBLOCK:
404 #endif
405                                 break;
406                         case -ECONNREFUSED:
407                                 /* TODO: Take an action if remove_conn fails. */
408                                 remove_conn(&cvec, i);
409                                 break;
410                         default:
411                                 if (err > 0)
412                                         break;
413
414                                 fprintf(stderr, "Unhandled error: %s.\n",
415                                         strerror(-err));
416                                 break;
417                         }
418                 }
419
420                 /* Maybe only sleep if nothing was found in this iter. */
421                 usleep(1000);
422         }
423
424         ret = 0;
425
426 fail:
427         for (i = 0; i < n_addrs; i++) {
428                 if (listenfds[i].fd > 0)
429                         close(listenfds[i].fd);
430         }
431
432         free(listenfds);
433         freeconnvector(&cvec);
434
```

```
435         /* Do we meed this NULL check? */
436         if (addrs)
437                 freeaddrinfo(addrs);
438
439         return ret;
440 }
441
442 static int load_crypto_key_from_file(const char *filepath)
443 {
444         int fd;
445         ssize_t ret;
446
447         fd = open(filepath, O_RDONLY);
448         if (fd < 0) {
449                 perror("open");
450                 return -1;
451         }
452
453         ret = read(fd, crypto_key, sizeof(crypto_key));
454
455         if (ret < 0) {
456                 perror("read");
457                 return -1;
458         }
459
460         if (ret != sizeof(crypto_key)) {
461                 fprintf(stderr, "Crypto key: Tried to read %zu bytes, "
462                         "but only received %zd.\n", sizeof(crypto_key), ret);
463                 return -1;
464         }
465
466         close(fd);
467         return 0;
468 }
469
470 int main(int argc, char **argv)
471 {
472         const char *addr = "localhost";
473
474         init_signal_handlers();
475
476         if (argc >= 2)
477                 addr = argv[1];
478
479         memcpy(crypto_key, progger_crypto_testkey, sizeof(crypto_key));
480
481         if (argc >= 3) {
482                 fprintf(stderr, "Loading crypto key from '%s'.\n", argv[2]);
483                 if (load_crypto_key_from_file(argv[2]) != 0) {
484                         fprintf(stderr, "Failed to load crypto key.\n");
485                         return 1;
486                 }
487         }
488
489         listen_on_addr(addr);
490
491         return 0;
492 }
```

A.29 drivers/net/progger/server/syscall-table.c

```
// SPDX-License-Identifier: GPL-2.0-only

#include "syscalls.h"

#include <stdint.h>

#include <progger/compiler.h>

/*
 * grep -E '^[0-9]' arch/x86/entry/syscalls/syscall_64.tbl | \
 *     awk '{print "\t[" $1 "] = \"" $3 "\","}'
 */
static const char *const syscalls[] = {
       [0] = "read",
       [1] = "write",
       [2] = "open",
       [3] = "close",
       [4] = "stat",
       [5] = "fstat",
       [6] = "lstat",
       [7] = "poll",
       [8] = "lseek",
       [9] = "mmap",
       [10] = "mprotect",
       [11] = "munmap",
       [12] = "brk",
       [13] = "rt_sigaction",
       [14] = "rt_sigprocmask",
       [15] = "rt_sigreturn",
       [16] = "ioctl",
       [17] = "pread64",
       [18] = "pwrite64",
       [19] = "readv",
       [20] = "writev",
       [21] = "access",
       [22] = "pipe",
       [23] = "select",
       [24] = "sched_yield",
       [25] = "mremap",
       [26] = "msync",
       [27] = "mincore",
       [28] = "madvise",
       [29] = "shmget",
       [30] = "shmat",
       [31] = "shmctl",
       [32] = "dup",
       [33] = "dup2",
       [34] = "pause",
       [35] = "nanosleep",
       [36] = "getitimer",
       [37] = "alarm",
       [38] = "setitimer",
       [39] = "getpid",
       [40] = "sendfile",
       [41] = "socket",
       [42] = "connect",
```

```
        [43] = "accept",
        [44] = "sendto",
        [45] = "recvfrom",
        [46] = "sendmsg",
        [47] = "recvmsg",
        [48] = "shutdown",
        [49] = "bind",
        [50] = "listen",
        [51] = "getsockname",
        [52] = "getpeername",
        [53] = "socketpair",
        [54] = "setsockopt",
        [55] = "getsockopt",
        [56] = "clone",
        [57] = "fork",
        [58] = "vfork",
        [59] = "execve",
        [60] = "exit",
        [61] = "wait4",
        [62] = "kill",
        [63] = "uname",
        [64] = "semget",
        [65] = "semop",
        [66] = "semctl",
        [67] = "shmdt",
        [68] = "msgget",
        [69] = "msgsnd",
        [70] = "msgrcv",
        [71] = "msgctl",
        [72] = "fcntl",
        [73] = "flock",
        [74] = "fsync",
        [75] = "fdatasync",
        [76] = "truncate",
        [77] = "ftruncate",
        [78] = "getdents",
        [79] = "getcwd",
        [80] = "chdir",
        [81] = "fchdir",
        [82] = "rename",
        [83] = "mkdir",
        [84] = "rmdir",
        [85] = "creat",
        [86] = "link",
        [87] = "unlink",
        [88] = "symlink",
        [89] = "readlink",
        [90] = "chmod",
        [91] = "fchmod",
        [92] = "chown",
        [93] = "fchown",
        [94] = "lchown",
        [95] = "umask",
        [96] = "gettimeofday",
        [97] = "getrlimit",
        [98] = "getrusage",
        [99] = "sysinfo",
        [100] = "times",
        [101] = "ptrace",
        [102] = "getuid",
```

```
117         [103] = "syslog",
118         [104] = "getgid",
119         [105] = "setuid",
120         [106] = "setgid",
121         [107] = "geteuid",
122         [108] = "getegid",
123         [109] = "setpgid",
124         [110] = "getppid",
125         [111] = "getpgrp",
126         [112] = "setsid",
127         [113] = "setreuid",
128         [114] = "setregid",
129         [115] = "getgroups",
130         [116] = "setgroups",
131         [117] = "setresuid",
132         [118] = "getresuid",
133         [119] = "setresgid",
134         [120] = "getresgid",
135         [121] = "getpgid",
136         [122] = "setfsuid",
137         [123] = "setfsgid",
138         [124] = "getsid",
139         [125] = "capget",
140         [126] = "capset",
141         [127] = "rt_sigpending",
142         [128] = "rt_sigtimedwait",
143         [129] = "rt_sigqueueinfo",
144         [130] = "rt_sigsuspend",
145         [131] = "sigaltstack",
146         [132] = "utime",
147         [133] = "mknod",
148         [134] = "uselib",
149         [135] = "personality",
150         [136] = "ustat",
151         [137] = "statfs",
152         [138] = "fstatfs",
153         [139] = "sysfs",
154         [140] = "getpriority",
155         [141] = "setpriority",
156         [142] = "sched_setparam",
157         [143] = "sched_getparam",
158         [144] = "sched_setscheduler",
159         [145] = "sched_getscheduler",
160         [146] = "sched_get_priority_max",
161         [147] = "sched_get_priority_min",
162         [148] = "sched_rr_get_interval",
163         [149] = "mlock",
164         [150] = "munlock",
165         [151] = "mlockall",
166         [152] = "munlockall",
167         [153] = "vhangup",
168         [154] = "modify_ldt",
169         [155] = "pivot_root",
170         [156] = "_sysctl",
171         [157] = "prctl",
172         [158] = "arch_prctl",
173         [159] = "adjtimex",
174         [160] = "setrlimit",
175         [161] = "chroot",
176         [162] = "sync",
```

```
177         [163] = "acct",
178         [164] = "settimeofday",
179         [165] = "mount",
180         [166] = "umount2",
181         [167] = "swapon",
182         [168] = "swapoff",
183         [169] = "reboot",
184         [170] = "sethostname",
185         [171] = "setdomainname",
186         [172] = "iopl",
187         [173] = "ioperm",
188         [174] = "create_module",
189         [175] = "init_module",
190         [176] = "delete_module",
191         [177] = "get_kernel_syms",
192         [178] = "query_module",
193         [179] = "quotactl",
194         [180] = "nfsservctl",
195         [181] = "getpmsg",
196         [182] = "putpmsg",
197         [183] = "afs_syscall",
198         [184] = "tuxcall",
199         [185] = "security",
200         [186] = "gettid",
201         [187] = "readahead",
202         [188] = "setxattr",
203         [189] = "lsetxattr",
204         [190] = "fsetxattr",
205         [191] = "getxattr",
206         [192] = "lgetxattr",
207         [193] = "fgetxattr",
208         [194] = "listxattr",
209         [195] = "llistxattr",
210         [196] = "flistxattr",
211         [197] = "removexattr",
212         [198] = "lremovexattr",
213         [199] = "fremovexattr",
214         [200] = "tkill",
215         [201] = "time",
216         [202] = "futex",
217         [203] = "sched_setaffinity",
218         [204] = "sched_getaffinity",
219         [205] = "set_thread_area",
220         [206] = "io_setup",
221         [207] = "io_destroy",
222         [208] = "io_getevents",
223         [209] = "io_submit",
224         [210] = "io_cancel",
225         [211] = "get_thread_area",
226         [212] = "lookup_dcookie",
227         [213] = "epoll_create",
228         [214] = "epoll_ctl_old",
229         [215] = "epoll_wait_old",
230         [216] = "remap_file_pages",
231         [217] = "getdents64",
232         [218] = "set_tid_address",
233         [219] = "restart_syscall",
234         [220] = "semtimedop",
235         [221] = "fadvise64",
236         [222] = "timer_create",
```

```
237         [223] = "timer_settime",
238         [224] = "timer_gettime",
239         [225] = "timer_getoverrun",
240         [226] = "timer_delete",
241         [227] = "clock_settime",
242         [228] = "clock_gettime",
243         [229] = "clock_getres",
244         [230] = "clock_nanosleep",
245         [231] = "exit_group",
246         [232] = "epoll_wait",
247         [233] = "epoll_ctl",
248         [234] = "tgkill",
249         [235] = "utimes",
250         [236] = "vserver",
251         [237] = "mbind",
252         [238] = "set_mempolicy",
253         [239] = "get_mempolicy",
254         [240] = "mq_open",
255         [241] = "mq_unlink",
256         [242] = "mq_timedsend",
257         [243] = "mq_timedreceive",
258         [244] = "mq_notify",
259         [245] = "mq_getsetattr",
260         [246] = "kexec_load",
261         [247] = "waitid",
262         [248] = "add_key",
263         [249] = "request_key",
264         [250] = "keyctl",
265         [251] = "ioprio_set",
266         [252] = "ioprio_get",
267         [253] = "inotify_init",
268         [254] = "inotify_add_watch",
269         [255] = "inotify_rm_watch",
270         [256] = "migrate_pages",
271         [257] = "openat",
272         [258] = "mkdirat",
273         [259] = "mknodat",
274         [260] = "fchownat",
275         [261] = "futimesat",
276         [262] = "newfstatat",
277         [263] = "unlinkat",
278         [264] = "renameat",
279         [265] = "linkat",
280         [266] = "symlinkat",
281         [267] = "readlinkat",
282         [268] = "fchmodat",
283         [269] = "faccessat",
284         [270] = "pselect6",
285         [271] = "ppoll",
286         [272] = "unshare",
287         [273] = "set_robust_list",
288         [274] = "get_robust_list",
289         [275] = "splice",
290         [276] = "tee",
291         [277] = "sync_file_range",
292         [278] = "vmsplice",
293         [279] = "move_pages",
294         [280] = "utimensat",
295         [281] = "epoll_pwait",
296         [282] = "signalfd",
```

```
297         [283] = "timerfd_create",
298         [284] = "eventfd",
299         [285] = "fallocate",
300         [286] = "timerfd_settime",
301         [287] = "timerfd_gettime",
302         [288] = "accept4",
303         [289] = "signalfd4",
304         [290] = "eventfd2",
305         [291] = "epoll_create1",
306         [292] = "dup3",
307         [293] = "pipe2",
308         [294] = "inotify_init1",
309         [295] = "preadv",
310         [296] = "pwritev",
311         [297] = "rt_tgsigqueueinfo",
312         [298] = "perf_event_open",
313         [299] = "recvmmsg",
314         [300] = "fanotify_init",
315         [301] = "fanotify_mark",
316         [302] = "prlimit64",
317         [303] = "name_to_handle_at",
318         [304] = "open_by_handle_at",
319         [305] = "clock_adjtime",
320         [306] = "syncfs",
321         [307] = "sendmmsg",
322         [308] = "setns",
323         [309] = "getcpu",
324         [310] = "process_vm_readv",
325         [311] = "process_vm_writev",
326         [312] = "kcmp",
327         [313] = "finit_module",
328         [314] = "sched_setattr",
329         [315] = "sched_getattr",
330         [316] = "renameat2",
331         [317] = "seccomp",
332         [318] = "getrandom",
333         [319] = "memfd_create",
334         [320] = "kexec_file_load",
335         [321] = "bpf",
336         [322] = "execveat",
337         [323] = "userfaultfd",
338         [324] = "membarrier",
339         [325] = "mlock2",
340         [326] = "copy_file_range",
341         [327] = "preadv2",
342         [328] = "pwritev2",
343         [329] = "pkey_mprotect",
344         [330] = "pkey_alloc",
345         [331] = "pkey_free",
346         [332] = "statx",
347         [333] = "io_pgetevents",
348         [334] = "rseq",
349         [424] = "pidfd_send_signal",
350         [425] = "io_uring_setup",
351         [426] = "io_uring_enter",
352         [427] = "io_uring_register",
353         [428] = "open_tree",
354         [429] = "move_mount",
355         [430] = "fsopen",
356         [431] = "fsconfig",
```

```
357         [432] = "fsmount",
358         [433] = "fspick",
359         [434] = "pidfd_open",
360         [435] = "clone3",
361         [437] = "openat2",
362         [438] = "pidfd_getfd",
363         [439] = "faccessat2",
364         [512] = "rt_sigaction",
365         [513] = "rt_sigreturn",
366         [514] = "ioctl",
367         [515] = "readv",
368         [516] = "writev",
369         [517] = "recvfrom",
370         [518] = "sendmsg",
371         [519] = "recvmsg",
372         [520] = "execve",
373         [521] = "ptrace",
374         [522] = "rt_sigpending",
375         [523] = "rt_sigtimedwait",
376         [524] = "rt_sigqueueinfo",
377         [525] = "sigaltstack",
378         [526] = "timer_create",
379         [527] = "mq_notify",
380         [528] = "kexec_load",
381         [529] = "waitid",
382         [530] = "set_robust_list",
383         [531] = "get_robust_list",
384         [532] = "vmsplice",
385         [533] = "move_pages",
386         [534] = "preadv",
387         [535] = "pwritev",
388         [536] = "rt_tgsigqueueinfo",
389         [537] = "recvmmsg",
390         [538] = "sendmmsg",
391         [539] = "process_vm_readv",
392         [540] = "process_vm_writev",
393         [541] = "setsockopt",
394         [542] = "getsockopt",
395         [543] = "io_setup",
396         [544] = "io_submit",
397         [545] = "execveat",
398         [546] = "preadv2",
399         [547] = "pwritev2",
400 };
401
402 const char *syscall_str_from_nr(uint32_t nr)
403 {
404         if (nr >= ARRAY_SIZE(syscalls))
405                 return "INVALID";
406
407         return syscalls[nr];
408 }
```

A.30 drivers/net/progger/server/syscalls.h

```
1 /* SPDX-License-Identifier: GPL-2.0-only */
2
```

```
3  #ifndef PROGGER_SERVER_SYSCALLS_H
4  #define PROGGER_SERVER_SYSCALLS_H
5
6  #include <stdint.h>
7
8  const char *syscall_str_from_nr(uint32_t nr);
9
10 #endif /* PROGGER_SERVER_SYSCALLS_H */
```

A.31 include/progger/compiler.h

```
1  /* SPDX-License-Identifier: GPL-2.0-only */
2
3  #ifndef PROGGER_COMMON_COMPILER_H
4  #define PROGGER_COMMON_COMPILER_H
5
6  #ifdef __KERNEL__
7  #include <linux/compiler_types.h>
8  #else /* __KERNEL__ */
9
10 #ifndef ARRAY_SIZE
11 #define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
12 #endif
13
14 #ifndef likely
15 #define likely(x)     __builtin_expect(!!(x), 1)
16 #endif
17
18 #ifndef unlikely
19 #define unlikely(x)   __builtin_expect(!!(x), 0)
20 #endif
21
22 #ifndef __packed
23 #define __packed      __attribute__((packed))
24 #endif
25
26 #ifndef __unused
27 #define __unused      __attribute__((unused))
28 #endif
29
30 #endif /* __KERNEL__ */
31
32 #endif /* PROGGER_COMMON_COMPILER_H */
```

A.32 include/progger/crypto.h

```
1  /* SPDX-License-Identifier: GPL-2.0-only */
2
3  #ifndef PROGGER_COMMON_CRYPTO_H
4  #define PROGGER_COMMON_CRYPTO_H
5
6  #ifdef __KERNEL__
7  #include <crypto/chacha20poly1305.h>
8  #else
```

```
#define CHACHA20POLY1305_KEY_SIZE   32
#define XCHACHA20POLY1305_NONCE_SIZE 24
#define CHACHA20POLY1305_AUTHTAG_SIZE 16
#endif /* __KERNEL__ */

#include <progger/types.h>

static const uint8_t progger_crypto_testkey[CHACHA20POLY1305_KEY_SIZE] = {
       0xca, 0xfe, 0x13, 0x37,
       0xca, 0xfe, 0x13, 0x37,
       0xca, 0xfe, 0x13, 0x37,
       0xca, 0xfe, 0x13, 0x37,
       0xca, 0xfe, 0x13, 0x37,
       0xca, 0xfe, 0x13, 0x37,
       0xca, 0xfe, 0x13, 0x37,
       0xca, 0xfe, 0x13, 0x37,
};

#endif /* PROGGER_COMMON_CRYPTO_H */
```

A.33 include/progger/net.h

```
/* SPDX-License-Identifier: GPL-2.0-only */

#ifndef PROGGER_COMMON_NET_H
#define PROGGER_COMMON_NET_H

#define PROGGER_SERVER_PORT  13753

#endif /* PROGGER_COMMON_NET_H */
```

A.34 include/progger/record.h

```
/* SPDX-License-Identifier: GPL-2.0-only */

#ifndef PROGGER_COMMON_RECORD_H
#define PROGGER_COMMON_RECORD_H

#include <progger/compiler.h>
#include <progger/types.h>

#ifdef __KERNEL__
#include <linux/build_bug.h>
#include <linux/stddef.h>
#define assert_field_size(type, field, size) \
       static_assert(sizeof_field(type, field) == size)
#else
#define assert_field_size(type, field, size)
#endif

#define SYSCALL_MAX_ARGS_X86_64 6

#define MAX_RECORD_STR_SIZE 4096
#define MAX_RECORD_SIZE (1024 + (SYSCALL_MAX_ARGS_X86_64 *
```

```
    MAX_RECORD_STR_SIZE))

#define TP_SRC_SYS_ENTER (1 << 1)
#define TP_SRC_SYS_EXIT (1 << 2)

#define PROGGER_RECORD_PADDING_ALIGN 32

struct record_ad {
	uint32_t len;
	uint64_t client_id;
} __packed;

struct record {
	uint32_t len;
	uint16_t id;
	uint16_t reserved;
	uint8_t data[];
} __packed;

struct record_syscall_x86_64 {
	uint8_t tp_src;
	uint8_t reserved;
	uint16_t nr;
	uint64_t ts;
	uint64_t ret;
	uint32_t pid;
	uint32_t uid;
	uint32_t euid;
	uint64_t args[SYSCALL_MAX_ARGS_X86_64];
	char strings[];
} __packed;

assert_field_size(struct record_syscall_x86_64, pid, sizeof(pid_t));
assert_field_size(struct record_syscall_x86_64, uid, sizeof(uid_t));
assert_field_size(struct record_syscall_x86_64, euid, sizeof(uid_t));

enum {
	RECORD_ENUM_MIN = 1024,
	RECORD_SYSCALL_X86_64 = 1025,
	RECORD_ENUM_MAX = 65535
};

static inline struct record *next_record(struct record *record)
{
	struct record *next = (struct record *)(((char *)record) + record->len);

	if (next->len == 0)
		return NULL;

	return next;
}

#endif /* PROGGER_COMMON_RECORD_H */
```

A.35 include/progger/types.h

```
/* SPDX-License-Identifier: GPL-2.0-only */
```

```
2
3  #ifndef PROGGER_COMMON_TYPES_H
4  #define PROGGER_COMMON_TYPES_H
5
6  #ifdef __KERNEL__
7  #include <linux/compiler.h>
8  #include <linux/types.h>
9  #include <linux/stddef.h>
10 #else
11 #include <stdbool.h>
12 #include <stddef.h>
13 #include <stdint.h>
14 #include <sys/types.h>
15 #include <asm/types.h>
16 #endif /* __KERNEL__ */
17
18 #endif /* PROGGER_COMMON_TYPES_H */
```

Appendix B

The `new-session` program

```
#define _GNU_SOURCE

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

void setsid_and_fork(void)
{
        pid_t new_sid;
        pid_t new_pid;

        new_sid = setsid();
        if (new_sid < 0) {
                perror("setsid");
                exit(1);
        }
        printf(" New session ID: %d.\n", new_sid);

        new_pid = fork();
        if (new_pid < 0) {
                perror("fork");
                exit(1);
        } else if (new_pid > 0) {
                printf("Child process ID: %d.\n", new_pid);
                _exit(0);
        }
}

int main(int argc, char **argv)
{
        if (geteuid() != 0) {
                fprintf(stderr, "Run this as root.\n");
                return 1;
        }

        printf("Orig. session ID: %d.\n", getsid(0));
        printf("Orig. process ID: %d.\n", getpid());

```

```
41          setsid_and_fork();
42          setsid_and_fork();
43
44          if (argc < 2)
45                  return 0;
46
47          printf("execvp(\"%s\", [", argv[1]);
48          for (int i = 1; i < argc; i++)
49                  printf("\"%s\", ", argv[i]);
50          printf("\b\b])\n");
51
52          execvp(argv[1], &argv[1]);
53          perror("execvp");
54
55          return 1;
56  }
```