Design & Architecture of Progger 3: A Low-Overhead, Tamper-Proof Provenance System

Tristan Corrick and Vimal Kumar(✉)[0000−0002−4955−3058]

School of Computing and Mathematical Sciences,
University of Waikato, Hamilton 3200, New Zealand
tristan@corrick.kiwi
vkumar@waikato.ac.nz

Abstract. This paper presents Progger 3, the latest version of the provenance collection tool, Progger. We outline the design goals for Progger 3 and describe in detail how the architecture achieves those goals. In contrast to previous versions of Progger, this version 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.

Keywords: Data Provenance; Accountability; Tamper-proof provenance; Security; Secure Provenance

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?", etc. [1].

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 generated.

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 latter approach was taken by Ko et. al. in Progger 1[3] and Progger 2. In this paper, we present the latest version of Progger, that we call Progger 3. Like its predecessors, Progger 3 also traces Linux system calls

in order to collect provenance. It addresses certain security and performance issues in Progger 1 and Progger 2. Progger 3 is designed to be truly kernel-only. The kernel-only mode is combined 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 means that the provenance can be transferred over the network, encrypted, providing confidentiality and integrity. Also, as user space can never access the cryptographic keys, it cannot generate false provenance records. Progger 3 was also 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.

2 Design goals

Progger 3 was designed with the following goals in mind. It is very important to understand that a strong motivation behind Progger 3's design is to prevent an untrusted user space from being able to maliciously impact Progger 3's operation.

- **A.** The provenance system is kernel-*only*, meaning that user space cannot alter the provenance system client's code, configuration, 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.

3 The Architecture of Progger 3

Progger 3 is designed as 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 system calls. 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.

It should be kept in mind that 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

[1] This achieves design goal J

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. This is because 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. Of course, process filtering can be done later by a program that processes the data collected by Progger 3.

While this causes 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 under their threat model, and weigh those advantages 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 run 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 Record format

Each message Progger 3 sends across the network contains a set of *records* that are encrypted and authenticated by XChaCha20-Poly1305 [8]. 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`.

4.1 Header format

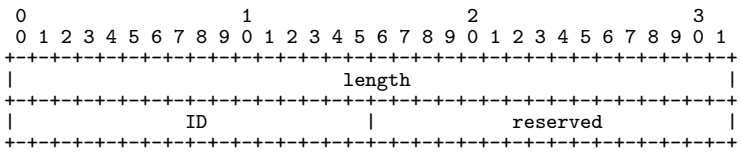


Fig. 1: The record header format

The record header format as set out in Fig. 1 has the following fields:

– length — The length in bytes of the whole record with 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.

4.2 Body format

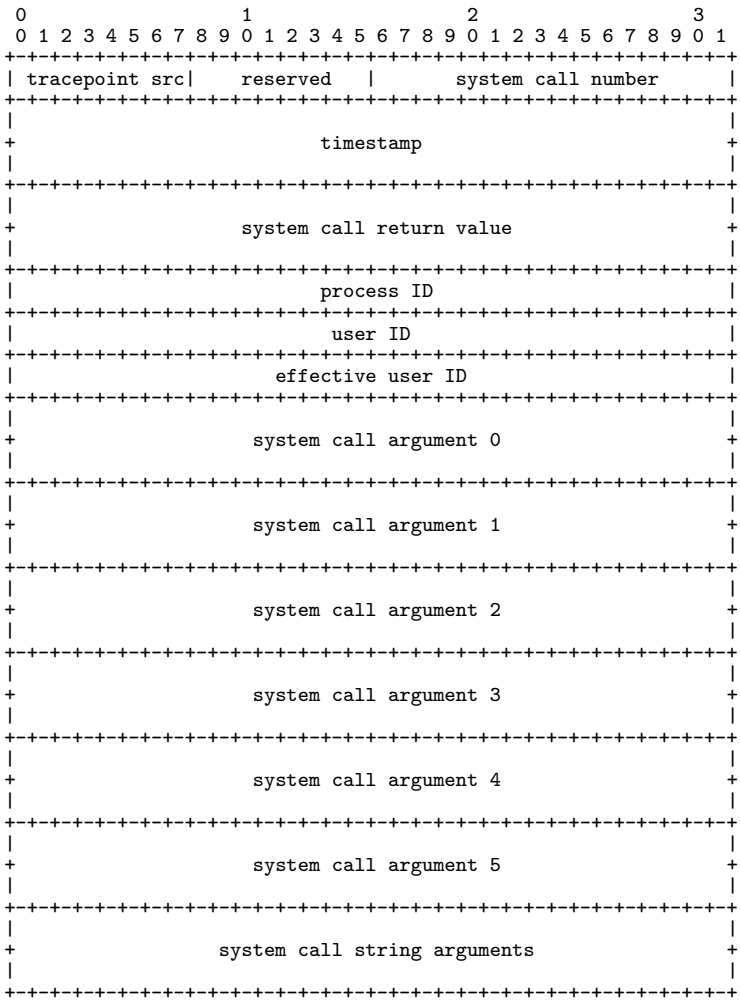


Fig. 2: The `RECORD_SYSCALL_X86_64` format

With 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 described in Fig. 2. 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). 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.
- system call number — The system call number from Linux. A 2-octet field.
- timestamp — A timestamp representing the number of nanoseconds since the system booted.
- 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 arguments 0 to 5.

This record notably 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 require 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 Server JSON output

Listing 1.1 provides a sample of the output of our Progger 3 server implementation when the system calls `openat`, and `setresuid` are being monitored.

Listing 1.1: 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": [ ] }
```

5 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 rest of this section describes how Progger 3 achieves its kernel-only mode of operation, i.e. design goal A.

5.1 Trusted kernels

Having the Progger 3 client run entirely as a kernel module is necessary for a kernel-only mode of operation, but it is not sufficient. To ensure kernel-only operation, the user must verify that the kernel itself cannot be tampered with; that is, the kernel can be trusted. Since Progger 3 has a secret key in kernel memory, the user must boot with the kernel argument `lockdown=confidentiality`. This will ensure that, 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. Without `lockdown` set any root user would be able to insert arbitrary kernel modules. Thus being able to probe for Progger 3's secret key, or even stopping the tasks Progger 3 runs. 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`.

Using a trusted kernel, however, 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 may 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 private 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 significant areas where it can be used.

5.2 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 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 may render the installation untrustworthy in some threat models. Being built-in to the kernel, which is a new feature in Progger 3, 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 achieving design goal H.

6 Cryptography

This Progger 3 logs include information such as the file names on a system, the time of each file access, as well as the programs being executed etc. Access to this level of detailed information can reveal to an adversary, in real-time or retroactively, what activities are, or might be taking place, on the system. Progger 3, therefore, uses 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.

6.1 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 [7]. Instead, Progger 3 uses a simpler approach based on XChaCha20-Poly1305 [8]

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 un-tampered for verification to succeed. The format of this message, can be seen in Fig. 3.

For security purposes, Progger 3 does not have its own implementation of any cryptographic algorithm instead relying on the Linux kernel's crypto library. XChaCha20-Poly1305 was chosen over ChaCha20-Poly1305 due to its 192-bit nonce as opposed to a 96-bit nonce in ChaCha20-Poly1305 [8]. 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 [6]. 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

[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 [5].

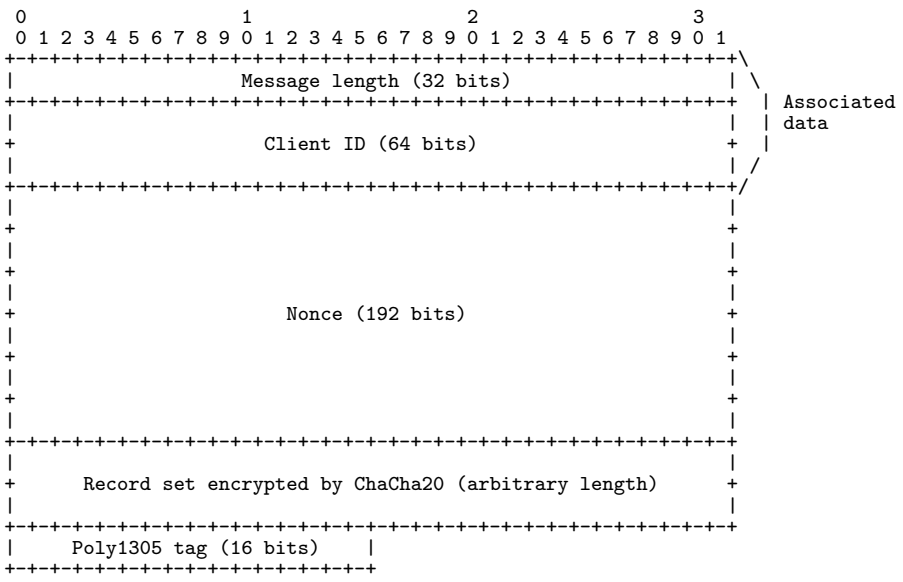


Fig. 3: Format of messages sent over the network by Progger 3

the final 64 bits. This construction is not security-affecting compared to using an entirely random nonce, or an entirely counter-based nonce [8]. 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 Fig. 3. Progger 3, ideally, should 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.

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 also 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. This assures that messages transferred by Progger 3 are done so with confidentiality and integrity and means that design goals C and D have been met.

6.2 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.

To prevent this from happening, Progger 3 uses the Trusted Platform Module (TPM) to seal the key when the system is provisioned. A TPM policy is used so that the key can only be unsealed when a chosen Platform Configuration Register (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.

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. Further, one could also seal the private key for client authentication using a TPM and use the Diffie–Hellman key exchange to generate the XChaCha20-Poly1305 key. While this would provide some security advantages, it would would increase complexity and a large amount of computation would need to be done in the kernel.

As the disadvantages were deemed to outweigh the advantages, it was not considered appropriate to utilise an ephemeral Diffie–Hellman key exchange.

7 Trusted Platform Module

As previously 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 [2].

7.1 TPM provisioning

When a system is provisioned with Progger 3, a unique symmetric key 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 [9].

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* [9]. 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.

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.

7.2 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 takes the following steps.

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.

7.3 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 makes it 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 achieves design goal B.

8 Traceability

While Progger 1 was only able to trace 32 system calls [3], and Progger 2 only supported 23, 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). This is because Progger 3 uses a single, simple function for handling every system call, while Progger 1 and Progger 2 both used 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 both the system call entry and exit tracepoints, and as such can trace any system call with a single function. It is to be noted that the callback function used in Progger 2 for the system call entry tracepoint still ended up calling individual functions for different system calls.

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. Thus, a single code path can deal with tracing any system call.

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.

9 Stability & Maintainability

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. No system instability has been observed with Progger 3 thus far. 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 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. Hence, design goal I is met.

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 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. Also, since Progger 3 uses a single code path to process system calls, it is straightforward to make changes to data that is gathered for each system call.

10 Performance improvements

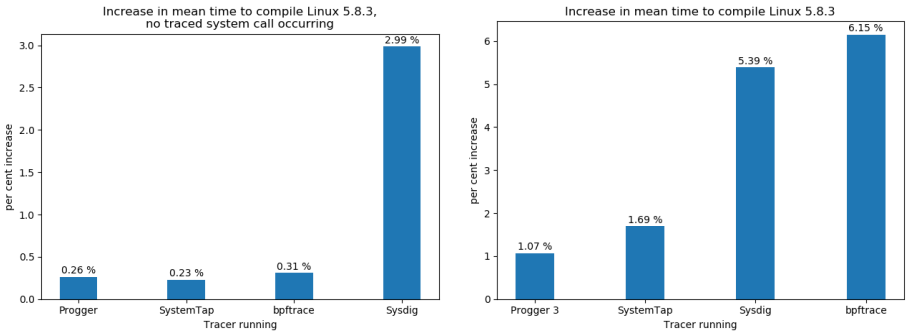
Progger 3 has significant performance improvements over Progger 1 and Progger 2. In this paper we focus on the steps taken to achieve the performance increase. Some benchmark results on the performance can be found in [2] .

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. Progger 3 minimises this overhead in order to achieve

[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.

performance improvements over its predecessors. Progger 3 makes use of a ring buffer and writes information to it when a system call executes. Any further processing, such as encryption and transferring the data over TCP, is done by a separate kernel task. 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 reduction in system performance is not too severe. We have ensured that the ring buffer implementation at the code level 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.

Thus, Progger 3 achieves efficiency by separating system call data collection from data processing and having code that works in an efficient manner. Progger 3 can be reasonably used in a wide range of workloads without reducing system performance to an unacceptable level. Therefore, design goal E is achieved. Below, we present our initial experiment of comparing Progger 3 with other tracer tools namely, Sysdig, SystemTap and bpftTrace. We compiled Linux 5.8.3 and observed the overhead caused by various system tracers. As can be seen in Fig. 4, Progger 3 incurs the least overhead when system calls are being traced while only being behind SystemTap when no system call tracing occurs.



(a) Linux compile test results (b) Linux compile test results

Fig. 4: Comparison of overhead with system tracers running

11 Conclusion

Progger 3 has an architecture with many parts, but still retains simplicity due to careful and purposeful architectural decisions. By ensuring kernel-only operation, along with the use of strong cryptography and a TPM, Progger 3 can provide tamper-proof logging of provenance both on the system running the Progger 3 client, and while the provenance is in transit to another system. Progger 3's message format 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 client. Furthermore, an efficiency-focused design means that Progger 3 can realistically be used under many workloads. The focus on simplicity results in increased usability that allows one to trace any system call with continuous, error-free operation.

Acknowledgments This work was supported by funding from STRATUS (https://stratus.org.nz), a science investment project funded by the New Zealand Ministry of Business, Innovation and Employment (MBIE). The authors would also like to acknowledge support from Prof. Ryan Ko and the team at Firstwatch for providing access to Progger 1 and 2.

References

1. Carata, L., Akoush, S., Balakrishnan, N., Bytheway, T., Sohan, R., Seltzer, M., Hopper, A.: A primer on provenance. Communications of the ACM **57**(5), 52–60 (2014). https://doi.org/10.1145/2596628
2. Corrick, T.: Progger 3: A low-overhead, tamper-proof provenance system. Master's thesis, The University of Waikato (2021). URL `https://hdl.handle.net/10289/14280`
3. Ko, R.K.L., Will, M.A.: Progger: An efficient, tamper-evident kernel-space logger for cloud data provenance tracking. In: IEEE 7th International Conference on Cloud Computing, pp. 881–889 (2014). https://doi.org/10.1109/CLOUD.2014.121
4. L. Torvalds *et al.*: 64-bit system call numbers and entry vectors. URL `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. L. Torvalds *et al.*: Kernel TLS. URL `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/networking/tls.rst?h=v5.8.18`
6. Nir, Y., Langley, A.: ChaCha20 and Poly1305 for IETF protocols. RFC 8439 (2018). URL `https://tools.ietf.org/html/rfc8439`
7. Rescorla, E.: The transport layer security (TLS) protocol version 1.3. RFC 8446 (2018). URL `https://tools.ietf.org/html/rfc8446`
8. S. Arciszewski: XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305. Tech. rep. URL `https://tools.ietf.org/html/draft-irtf-cfrg-xchacha-03`
9. Trusted Computing Group: Trusted Platform Module Library. URL `https://trustedcomputinggroup.org/resource/tpm-library-specification/`. Family "2.0", Rev. 01.59