Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS EU 2025 - Selected Papers from the 12th Annual Digital Forensics Research Conference Europe

When is logging sufficient? — Tracking event causality for improved forensic analysis and correlation

Johannes Olegård [*], Stefan Axelsson, Yuhong Li

Department of Computer and System Sciences, Stockholm University, Borgarfjordsgatan 12, Kista, 16407, Sweden

A R T I C L E I N F O

Keywords:
Provenance graph
Logging
Digital forensics
Anti-anti-forensics
Event-reconstruction

A B S T R A C T

It is generally agreed that logs are necessary for understanding cyberattacks post-incident. However, little is known about what specific information logs should contain to be forensically helpful. This uncertainty, combined with the fact that conventional logs are often not designed with security in mind, often results in logs with too much or too little information. Events in one log are also often challenging to correlate with events in other logs. Most previous research has focused on preserving, filtering, and interpreting logs, rather than addressing what should be logged in the first place. This paper explores logging sufficiency through the lens of Digital Forensic Readiness, and highlights the absence of *causal information* in conventional logs. To address this gap, we propose a novel logging system leveraging "gretel numbers" to track causal information—such as attacker movement—across multiple applications in a tamper-resistant manner. A prototype, implemented using the Extended Berkeley Packet Filter (EBPF) and an Nginx web server, shows that causality tracking imposes minimal resource overhead, though log size management remains critical for scalability.

1. Introduction

The question of how IT systems should be designed, in terms of logging, to help in forensic investigations of cyberattacks has received little attention in the research literature. It is generally agreed, however, that proper security logging is important, as evidenced by logging requirements in security standards like ISO 27001 (International Organization for Standardization, 2013) (section A.12.4) and NIST SP 800-92 (Kent and Souppaya, 2006), though these standards offer minimal guidance on what specific information should be logged.

The vast majority of research on this topic has been dedicated to the filtration and fuzzy correlation of conventional logs. These efforts aim to save storage space, identify case-relevant log entries, and combine logs from disparate systems. Even so, a significant amount of time is still required from forensic investigators to analyze the resulting refined logs and establish their validity in court. This suggests that conventional logs are imprecise, information-sparse, and not optimized for forensic purposes (cf. Fig. 1). Few studies question whether useful information is being produced in the first place and whether useful information exists that could be logged but is not currently logged. As Barse and Jonsson (2004) noted: "it may come as a surprise to the uninitiated that even after 20 years since the birth of IDS, it is still not known what kind of log data that are needed to detect different types of intrusions and attacks", and, while improvements have been made, we argue that this observation remains largely true today in the broader context of forensics purposes beyond Intrusion Detection Systems (IDS).

The research literature lacks a theoretical foundation for good logging practices (Azahari and Balzarotti, 2024), and in response, this paper makes three contributions:

- Introducing the *answerability of forensic questions* as a metric for evaluating log sufficiency.
- Identifying *causal information* as being crucial but absent from conventional logs.
- Proposing and evaluating the use of "gretel numbers" for tracking causal information in logs.

2. Related work

Early authors have noted the inherent limitations of logs, including: the trade-off between storage requirements and log accuracy (Bishop, 1990), and the skepticism surrounding the admissibility of logs in courts (Kenneally, 2004). These problems will likely remain for the foreseeable future, and this paper attempts to work within these limitations.

* Corresponding author. Department of Computer and System Sciences, Stockholm University, Sweden.
E-mail address: johannes.olegard@dsv.su.se (J. Olegård).

https://doi.org/10.1016/j.fsidi.2025.301877

Available online 24 March 2025

2666-2817/© 2025 The Author(s). Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

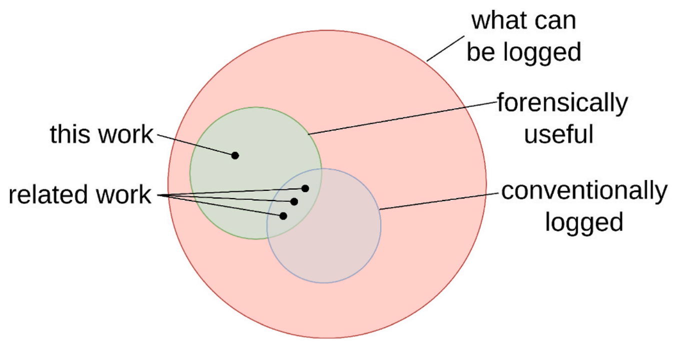


Fig. 1. Venn-diagram of loggable information of the problem domain of this paper: forensically useful information missing from conventional logs, contrasted with related work that extracts useful information from noisy conventional logs.

Other authors have used state machine models to reason about state reconstruction using logs (Gladyshev and Patel, 2004; Bishop, 1990; Juma et al., 2020). However, a drawback of this approach is the difficulty of constructing state machines of real systems.

Previous work has also addressed the storage and cryptographic preservation of logs (Schneier and Kelsey, 1999; Bhandary et al., 2020), as well as the standardization of log formats (Mitre Corporation, 2012). Note that such standards rarely prescribe what information should be contained in logs, only the format. For example, Mitre CEE (Mitre Corporation, 2012) mentions the need for logging requirements as part of the log management lifecycle, but does not go into specifics.

Threat modeling has been proposed as a method for determining logging requirements (Peisert et al., 2007; Rivera-Ortiz and Pasquale, 2020). However, little prescriptive guidance (beyond the use of their method) is provided, such as specific recommendations or detailed common log requirements.

Most research in this area has focused on filtering and refining existing logs, by means of forensic experiments to identify indicators of compromise (Barse and Jonsson, 2004), and techniques broadly summarized as "data mining" (King and Chen, 2005; Lee et al., 2013; Hossain et al., 2018; Michael et al., 2020; Goel et al., 2008). Before analysis, the data mining approaches often consolidate disparate logs into a unified data format, such as *provenance graphs* (King and Chen, 2005) (which include *evidence graphs* (Wang and Daniels, 2005)). Although provenance graphs are similar to those proposed in Section 5, they often lack precise granularity for correlation, and do not have the anti-tampering properties proposed there. Provenance graphs typically represent processes and objects as nodes, and time-stamped events as edges. In contrast, the proposed solution in this paper (Section 5) uses nodes to represent events and edges to represent causality relationships. While Lee et al. (2013) proposed subdividing each process into "units" by strategically inserting logging points through binary analysis, they still rely on system call logs—an approach which does not address the multi-host correlation problem.

3. Similar solutions

The idea of causality tracking using gretel numbers (Section 5) introduced in this paper, is similar to *distributed tracing* (OpenTelemetry Community, 2024), which is a method of combining traces from multiple networked applications into a single trace. A *trace* (OpenTelemetry Community, 2024) is a tree data structure describing what code has been executed by an application (each tree node typically represents a subroutine execution). A trace might, e.g., be a stack trace (when reporting a program crash) or a *flame graph* (Gregg, 2013) (for visualizing performance bottlenecks). *Context propagation* (OpenTelemetry Community, 2024) is linkage information used to combine traces and must be sent from the calling process to the called process. For example, if process A makes a remote procedure call to process B and B crashes, then the resulting distributed stack trace would consist of the A-trace with the B-trace attached as a subtree. The context propagation information in this scenario might consist of the parent-node ID (in A-trace), where the B-trace root should be attached. Gretel numbers, when transferred using messages, work similarly to context propagation information, thus, gretel numbers can be seen as combining distributed tracing with provenance graphs (in a tamper-resistant way), which (to our knowledge) has not been done before.

Context propagation has been standardized in the HTTP protocol (W3C, 2021), but most protocols (e.g., binary database protocols and system calls) do not support context propagation. While the *message IDs* (Resnick, 2001) in the email message format could be used to implement context propagation in Mail Transfer Agents (MTAs, e.g., SENDMAIL[1] and POSTFIX[2]), these IDs are normally set once by the Message Submission Agent (MSA) and then sent as-is through the chain of MTAs—effectively making the MTAs invisible in the distributed tracing system. Context information should be generated anew on each hop to prevent this. In comparison, the solution in Section 5 has a smaller intended scope (a single website/domain/system, rather than all email servers on the internet) and aims to be more extensive by spanning all protocols (not just email/SMTP), all system calls, and all messages (not just forwarded as-is).

Distributed tracing is not typically used for security (it is mostly intended for debugging and performance analysis), but previous work has proposed its use in anomaly detection (Jacob et al., 2021; Qiu et al., 2022). Previous non-security work (Shen et al., 2023) has used kernel network information to infer (which may be potentially unsuitable in a security context) process-internal causality information, in order to avoid the need to implement context propagation and instrumentation.

Prominent distributed tracing systems include tools such as OpenTelemetry[3] and Jaeger[4]. These tools typically consist of three core components: a centralized storage server, various *collection agents*, and a visualization tool. Collection agents might take the form of customizable software libraries (e.g., the OpenTelemetry client library for Python) or as plug-and-play-like modules (e.g., the OpenTelemetry module for Nginx). For the Proof-of-Concept presented in this paper (Section 6), the decision was made to build the system from adapting an existing distributed tracing tool. This approach was chosen because current tools lack crucial features that would necessitate extensive rewrites. Notably:

1. There is a lack of agents for certain software (e.g., the Linux kernel),
2. The agents lack support for certain functionality (e.g., the ability to track system calls),
3. Excessive details provided by some agents (e.g., entire stack traces),
4. Insufficient details provided by other (e.g., metrics-only data),
5. Lack of support for context propagation in system calls and binary database protocols, and
6. Constraints imposed by the trace (aka. "Span") tree-datastructure, which complicates some certain forms of causality tracking (e.g., response-flows and "stored" causality (e.g., in INODES).

As a result, existing systems would primarily be useful for storage and transfer to this storage—functions that are little useful in evaluating (Section 7) the proposed concept (Section 5).

4. Forensic Readiness for systems

Forensic Readiness (FR) has many definitions in the research

[1] https://www.proofpoint.com/us/products/email-protection/open-source-email-solution.
[2] https://postfix.org/.
[3] https://opentelemetry.io.
[4] https://jaegertracing.io.

literature, ranging from an organization's ability to conduct forensic investigations (Pangalos et al., 2010) to the properties of a system that allow investigation (Pasquale et al., 2018). While *log sufficiency* would fall in the latter domain, some argue (Daubner et al., 2024) that this research is often too high-level to be useful. Therefore, this paper proposes a narrower definition: *a system is forensically ready with respect to a set of forensic questions*. That is, if future questions, for some hypothetical investigation, can be predicted ahead of time, then the system under (future) investigation can be designed to facilitate accurate and reliable answers to those questions.

Thus, the question of "When is logging sufficient?" has been broken down by one level of abstraction, and what remains to be determined is: what are the common questions in cybersecurity investigations, and what logging-information would they require? This depends on the system, e.g., a social media network, an online bank, and a water treatment plant would all have questions in common, but also domain-specific questions. Furthermore, determining a complete list of questions (like those in (Goel et al., 2008)) would be equivalent to predicting the future and likely impossible—but through iteration, it is possible to continuously improve over time, cf. risk management methodology (International Organization for Standardization, 2013; Daubner and Matulevičius, 2021). More research is still required to survey what the common forensic questions are.

5. Causality tracking using gretel numbers

In the process of analyzing common forensic questions, we observed that logs often lack causal information, making correlating logs from different applications challenging. Therefore, such information is fundamental to answering most forensic questions in systems of multiple applications. See e.g., Fig. 2 where an Nginx access log is analyzed using the who-what-when-where-why-how (5W1H) questions (sometimes used as a checklist in forensic analysis), where causal information might fall under "Why" and "How". To address this gap, we introduce the concept of "gretel numbers," to enhance log correlation by embedding causal information. The rest of this section delves into the details of this concept.

Log entries describe events, and the causality of an event explains *why* that event occurred (which pertains to *How* in 5W1H, since *Why* describes human intentions). We make the case that most events (logged or otherwise) describe SENT and RECEIVED messages (cf. Figs. 3 and 4). The reason a message was RECEIVED is that it was SENT, and the reason it was SENT is usually caused by the processing of another RECEIVED message. This model captures not only the communication between applications but also: function calls between source code modules and system calls from a process to the operating system (OS) kernel.

Most modern systems consist of (non-human) interactive entities (e.g., containerized Linux processes) organized in a layered structure, as illustrated by the website in Fig. 3. The figure shows the layered processing of a single external message, which forms a graph. Each external message forms its own graph in this way, and if two external messages are processed the same way, their graphs will be isomorphic. In contrast, Fig. 4 shows the same scenario, but described using logged events (B1 has SENT-event SB1 and RECEIVED-event RB1). The reason message C1 was RECEIVED by DB (event RC1) is because API SERVER SENT message C1 (event SC1), which is because it RECEIVED message B1 (event RB1), and API SERVER is programmed to use C2 to construct B2.

The exact modeling of causality should be adapted to the needs of future investigations. For example, Fig. 5 models the same events as Fig. 4, but more accurately describes the control flow. Similarly, events can be inserted between two existing events to provide finer-grained logging, or merged or removed to save disk space (since each event results in a log entry).

Table 1 illustrates the log generated by WEB SERVER (in Fig. 5), which forms an inverse adjacency list of the WEB SERVER-part of the graph (effectively pointing back into the sender's log like a linked list). The *Metadata*-field represents data that would conventionally be logged (pertinent details from message A1). For distinction, each event must be assigned a unique identifier (ID) which, for clarity, will be referred to as the event's *gretel number* (after the Hansel and Gretel fairy tale) and similarly, Figs. 4 and 5 exemplify *gretel graphs*. In Table 1, the event-names from Fig. 4 (e.g., RA1) symbolize concrete gretel number (e.g., which in an implementation might be a number like: 12345). Note that Table 1 references SB2, which did not occur inside WEB SERVER, and this is only possible if message B2 contains gretel number SB2 (as part of the message metadata). Note also that each application must independently generate its gretel numbers, and this is explained in more detail in Section 6.

The correlation of logged events is achieved by merging the local gretel graphs from all logs, thereby reconstructing the complete gretel graph. For example, if the *Metadata* of log entry RC1 shows an SQL-injection attack, the combined gretel graph could be used to find the log entry RA1, where the *Metadata* would include the external IP address of the attacker. At that point, the investigator no longer needs the gretel graph and can focus on the *Metadata*-field of the respective log entries. Unlike in conventional logging, no fuzzy correlation (e.g., based on timestamps) is necessary, which makes the forensic analysis quicker and more accurate. Furthermore, fuzzy correlation is often critically reliant on time synchronization using e.g., the Network Time Protocol (NTP). In contrast, correlation based on gretel numbers works in the face of unsynchronized clocks and degraded timekeeping.

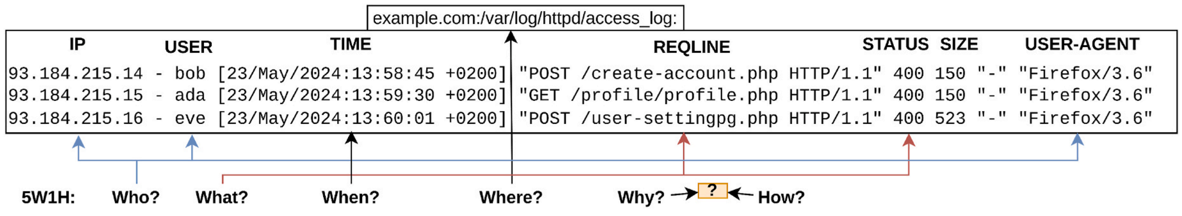
Besides correlation, gretel numbers can also detect log tampering in certain scenarios. The *ID* and *Predecessor IDs* fields in Table 1 roughly correspond to *credit* and *debit* in accounting (McClung, 1913) (tracking transactions between accounts), Due to the distributed nature of gretel numbers, and the slight overlap between gretel graphs, each application bears "witness" the actions of the systems it interacts with (similarly to how financial records of one company can inform on the malpractices of another company). For example, if an attacker achieves remote code execution in API SERVER (e.g., using a buffer overflow vulnerability), they can tamper with the API SERVER log, but they cannot (necessarily) tamper with the logs of the other servers. If the attacker deletes RB1, they cannot delete SB1 without also gaining access to the WEB SERVER (which is more work and will take additional time), and this deletion creates a detectable gap in the gretel graph. The attacker could omit or fake SC1 when sending C1, thus manipulating RC1—but doing so would produce another (non-isometric) graph which could be detected using anomaly detection (similar to how *credit* and *debit* should sum to zero in a balanced book). Implementing such an anomaly detection algorithm is not explored in this paper, but we note the potential here. Note especially how gretel numbers can extend to system calls logs, allowing the OS to act as (just) another server/witness (cf. related work on provenance graphs). Note also that the degree of redundancy and anti-tampering offered by gretel numbers is dependent on the isolation between "messaging entities" (i.e., the granularity chosen for *applications* in the system). If an attacker exploits a buffer overflow vulnerability in a Linux process, any logging performed by any part of that process (and any memory-sharing process) is susceptible to tampering. Thus, the process is a more suitable granularity for logging than individual source code modules (except to describe internal intricacies, such as when a cache hit substitutes a backend request).

6. Implementation

To test the practical feasibility of using gretel numbers, a Proof-Of-Concept (POC) was built[5], as illustrated in Fig. 6. The POC implementation consisted primarily of a customized Nginx web server[6], using

[5] available at: https://github.com/jesajx/gretel.

[6] https://nginx.org.



```
93.184.215.14 - bob [23/May/2024:13:58:45 +0200] "POST /create-account.php HTTP/1.1" 400 150 "-" "Firefox/3.6"
93.184.215.15 - ada [23/May/2024:13:59:30 +0200] "GET /profile/profile.php HTTP/1.1" 400 150 "-" "Firefox/3.6"
93.184.215.16 - eve [23/May/2024:13:60:01 +0200] "POST /user-settingpg.php HTTP/1.1" 400 523 "-" "Firefox/3.6"
```

Fig. 2. An example Nginx access log explained by mapping 5W1H to different fields (e.g., “When?” to *timestamp*, and “Who?” to *IP* and *user*.

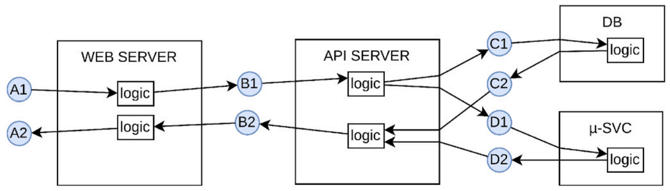


Fig. 3. Example logging scenario where an external message (A1) is processed by a website using internal message-passing. Processing spans four applications, organized in three layers: WEB SERVER, API SERVER, and lastly: DB and μ-SVC.

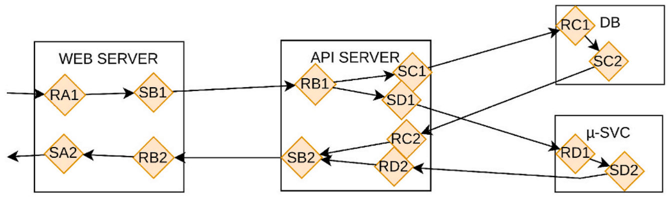


Fig. 4. The Fig. 3 message processing scenario from the perspective of logged events (nodes) and causality (edges): each SENT-event (Sxx) causes the corresponding RECEIVED-event (Rxx), which in turn causes other SENT-events (Syy).

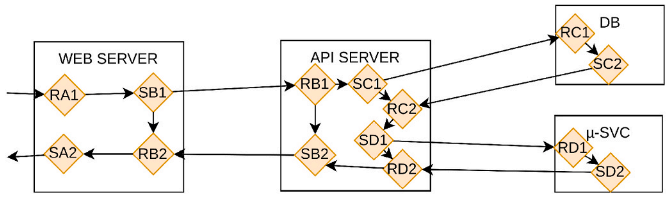


Fig. 5. Alternate modeling of Fig. 4 that more accurately models the causality (edges) of events by their sequential order in the control flow (in WEB SERVER and API SERVER applications, respectively).

Table 1
The log produced by WEB SERVER in Fig. 5, where each log entry is assigned a unique *ID*, and contains *Predecessor IDs* referencing other log entries (some in remote logs).

ID	Predecessor IDs	Metadata
RA1	(none)	…
SB1	RA1	…
RB2	SB1,SB2	…
SA2	RB2	…

build settings derived from Arch Linux[7], and an Extended Berkley Packet Filter (EBPF) module[8].

Nginx was used in the setup of two Docker[9] containers, each built

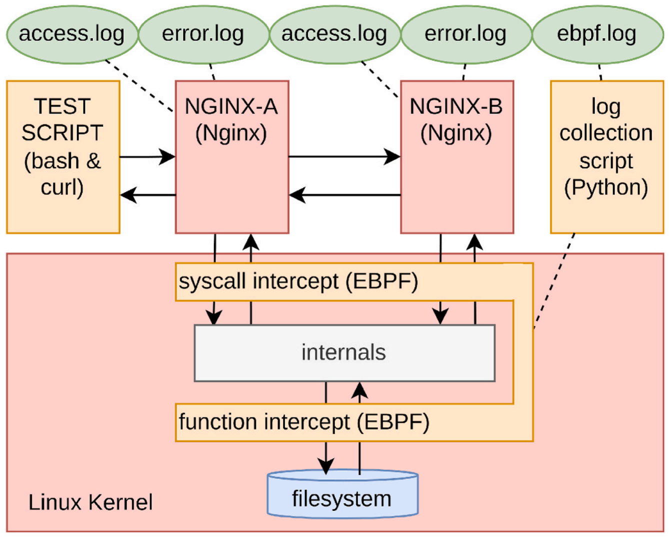


Fig. 6. Proof-Of-Concept architecture showing the communication between four main applications: TEST SCRIPT, NGINX-A, NGINX-B, and the Linux kernel—and various details (message interception and application–log associations).

from the same Nginx source code, to simulate a two-layer website architecture. The first instance (NGINX-A) was configured to forward specific requests to the second instance (NGINX-B), analogous to the API SERVER and DB configuration demonstrated in earlier examples. This reuse of code saved development time while still demonstrating the concept of gretel number.

EBPF is a Linux kernel feature for monitoring low-level events by executing custom code directly inside the kernel. The primary advantage of EBPF is that it achieves this securely and efficiently, by running with restricted capabilities. EBPF expands the functionality of an earlier project (that could only filter network packets) to cover additional use cases and kernel internals. EBPF was used in the POC to track and log gretel numbers through system calls made by the two Nginx instances. The POC EBPF module was written in the C programming language and compiled into EBPF bytecode (instead of regular x86-64 machine code). The bytecode is uploaded to the kernel via the EBPF system call and executed in a lightweight virtual machine. An EBPF module consists of callbacks, or “probes”, that are hooked to specific parts of the kernel. In the POC, the EBPF module registered callbacks for system calls entry and exit, as well as for select file-system-handling functions. The POC EBPF-module was managed using a Python script that used the BCC Toolkit[10] to handle compilation, upload, and logging.

The implementation involved four key aspects:

1. Nginx internals,
2. kernel internals,
3. Context propagation in all messages, and
4. a scheme for constructing gretel numbers.

[7] https://archlinux.org/packages/extra/x86_64/nginx.
[8] https://ebpf.io.
[9] https://docker.com.

[10] https://github.com/iovisor/bcc.

Since the focus of this paper is to improve logging practices, the need for such modifications should not be seen as a limitation or drawback. This stands in contrast to static analysis approaches, like that in (Lee et al., 2013), where logging is instrumented automatically to minimize developer intervention. We believe that logging must be treated as a "first-class citizen" in software development to achieve its full potential and to be explainable in court.

Nginx (1) was modified to inject and extract gretel numbers in messages and logs. For simplicity, HTTP/1.0 was used as the communication protocol (3), and a custom HTTP header was used to transmit gretel numbers (although a standardized context propagation header exists (W3C, 2021)). Tracking the gretel numbers between extraction and injection points primarily involved introducing a variable in each "thread". Note that Nginx uses the event reactor design pattern, which will not be explained here. This implementation aligns more closely with the control-flow-oriented approach depicted in Fig. 5 than the purely message-mapping approach in Fig. 4. Each thread handled forking, merging, and insertion of new gretel numbers by updating the variable in place. As a result, if each received message is processed by one new thread, the program maintains one variable per active message, alongside temporary text buffers for messages and logs. This per-message overhead is minimal in terms of memory consumption and processing. Note, however, that Nginx handles reading and writing separately, resulting in two "threads" per request. The local gretel graph was written in full to the Nginx error log, and the access logs only contained a subset of gretel numbers (corresponding to events analogous to SA2 in Fig. 4, and mirroring the log format in Table 1). For the experiment, additional logging points were added to the error log (but not the access log), to facilitate a comparison of log sizes (Section 7).

To facilitate the reader's ability to try the POC for themselves, EBPF (2) was chosen despite kernel modification being the simpler approach. The increased difficulty arises because EBPF, for security reasons, cannot modify the result of a system call, which prevents gretel numbers from being returned to Nginx from EBPF. The POC circumvents this limitation by sending both the Nginx-side SENT and RECEIVED gretel numbers to the kernel, where the kernel handles logging of the system-call-related edges. Furthermore, instead of adding a parameter to every system call—an approach that would break compatibility—gretel numbers are transmitted (3) to the kernel using the PRCTL system call (which is normally used to set/get miscellaneous process settings). An invalid PRCTL "OP-code" is used intentionally to make the Linux kernel ignore the call while still allowing EBPF to read the arguments. On the Nginx side (1), gretel numbers are not generated for every system call, and instead PRCTL is only called during context-switches between "threads" (i.e., resuming a paused thread) or when the active gretel number in the current thread changes—effectively the Nginx-side system call SENT-events are merged into the active thread gretel number. The EBPF stores the last reported gretel numbers for each running process (native thread).

Besides system calls, EBPF (2) is also used to store a gretel number for each (in-memory) INODE. When a process writes to an INODE, this can be seen as a "delayed" message that is delivered when another process reads from the INODE. Additional kernel internals could similarly be tracked, but the POC was limited to INODES for demonstration purposes.

Since the PRCTL system call can send at most 256 bits (four unsigned 64-bit numbers), the POC implementation used this size for all gretel numbers. While this size is significantly larger than necessary, it serves to demonstrate the "worst-case scenario". In practice, the gretel number scheme (4) should be adapted to the forensic requirements of the system under (future) investigation. Determining the *best* scheme is beyond the scope of this paper. However, for completeness, the basic POC gretel number scheme is outlined here. Each gretel number is subdivided into two components: a *location* and an *arbitrary but unique number*. The latter could be a random number, a counter, or a timestamp, with random numbers being the most common in the POC. The *location* uniquely represents the entity that generated the gretel number, typically consisting of an *APP-ID*. Optionally, the *location* can encode metadata about the event, such as type and or the specific source code line (e.g., print-statement) that generated the number. There are four applications in the POC: NGINX-A, NGINX-B, the TEST SCRIPT, and EBPF (with an additional EBPF instance required if the Nginx-servers run on separate hosts). Additionally, a placeholder would be needed for the "invalid application", bringing the total to five applications Thus, 3 bits are needed to represent *location*. However, for simplicity, the POC uses a multiple of 64 bit numbers to represent location. It is worth noting that in scenarios where an application (e.g., a single-process Docker container) operates as a singleton (i.e., at most one instance exists at a time), additional bits are unnecessary to differentiate between restarts—provided no *arbitrary but unique number* is reused. In a load-balancing scenario (e.g., multiple parallel copies of NGINX-A), it is generally better to assign each load-balancing "slot" is own *APP-ID*. In large-scale systems, a hierarchical *location* scheme, akin to IP subnetting, may be required to manage complexity.

In addition to the POC, a short Python script was used to translate the resulting logs into a graph file, enabling visualization in Gephi[11]. Fig. 7 presents the Gephi-exported image of an example gretel graph. The graph's nodes are color-coded as follows: blue nodes represent NGINX-A events (left side), green nodes represent NGINX-B events (right side), red nodes represent EBPF events in red (scattered throughout), and magenta nodes represent the TEST SCRIPT (bottom center).

The top section of the graph captures the Nginx startup process, while the bottom section shows iterations of the respective Nginx event handling loops, culminating in the processing of an HTTP request. Fig. 8 zooms in on this request-interaction (bottom center) of Fig. 7. The numbers highlight the sequence of events:

1. TEST SCRIPT sends a request to NGINX-A using curl,[12],
2. NGINX-A forwards the request to NGINX-B,
3. NGINX-B responds to NGINX-A, and
4. NGINX-A responds to TEST SCRIPT.

Finally, while the use of a centralized storage solution would be

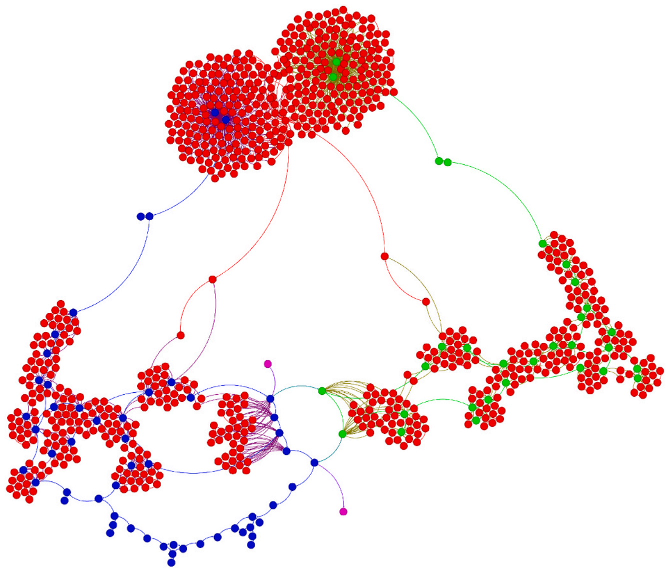


Fig. 7. POC-produced example gretel graph, including Nginx startup (top) and the step-by-step processing (bottom) of an HTTP request (magenta).

[11] https://gephi.github.io.
[12] https://curl.se.

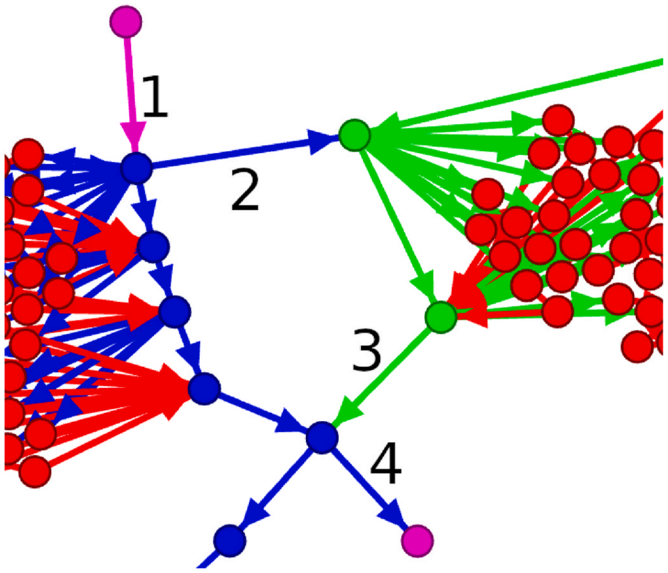


Fig. 8. Partial gretel graph showing the processing of an external request (magenta) by NGINX-A (blue), which passes through the backend NGINX-B. Nodes represent log entries, and the edges indicate causality. (For interpretation of the references to color in this figure legend, the reader is referred to the Web version of this article.)

expected to preserve logs containing gretel numbers—thereby protecting older logs from tampering—no such solution was included in the Proof-Of-Concept. Since gretel numbers are simply additional metadata appended to existing log entries, any standard log storage solution can accommodate them, including distributed redundant solutions. As a result, storage solutions were not further explored in this paper. It should be noted, however, that a gretel-number-aware storage solution could be beneficial for analysis (parsing gretel number schemes) and anti-tampering (gretel number spoofing). It is assumed that a storage solution would handle the authentication of each log-sending application. The credentials used for this authentication should, however, be assumed to be compromised at the same time as the respective application. Therefore, when an application mentions a specific gretel number in its log, this should be seen as a "claim" and scrutinized as such during forensic analysis.

7. Evaluation

To test the performance of the POC, a simple experiment was set up:

1. The POC was started and brought to a steady state.
2. Once stabilized, the experiment began, consisting of 10 thousand sequential requests sent from TEST SCRIPT to NGINX-B (via NGINX-A).
3. This experiment was repeated 10 times, under two conditions: with gretel numbers enabled ("with-gretel") and disabled ("without-gretel"). As a result, a total of 20 experiments was conducted.

For the "without-gretel" configuration, logging points were replaced using C-macros to exclude gretel numbers. The gretel C-functions remained in the resulting binary, but were never called. Furthermore, the EBPF module was not run during the "without-gretel" experiments.

It is also important to note that, unlike in Fig. 7, the logging of individual system calls was disabled for this experiment (i.e., removing most red nodes). Instead, only INODE-related events and process startup-related events were logged by the EBPF module. This choice was made to improve practicality and fairness, as logging every system call is not feasible in the real world.

The experiment used a desktop PC with an Intel Core i7-3770 CPU (3.40 GHz) CPU, 12 GiB RAM, a Solid State Drive (SSD) for OS files, and a Hard Disk Drive (HDD) for experiment files. The PC was running Arch Linux Kernel version 6.9.7-arch1-1. Little else was running on the PC during the experiments.

The observed variables for the experiment included: the current time (clock), user CPU time, system CPU times, and RAM usage. For each docker container CPU times and RAM usage were recorded at the start and end of the experiment, using values from the/SYS filesystem. Note that the system CPU time includes the time spent in the EBPF probes. Additionally, the size of the produced logs was measured after the experiment.

The results are presented in Table 2, with each cell formatted as $\mu \pm \sigma$, w consisting of the mean (μ) and Bessel-corrected standard deviation (σ). The *Duration* refers to the total time of the experiment. The *User Time* and *Sys. Time* are each combined CPU time of both NGINX-A and NGINX-B. While the doubling in CPU time may seem alarming, the time spent computing is dwarfed by the time waiting on input/output (cf. *Duration*). This simply indicates that Nginx is spending less time idle. Note also that the *test script* was considered *external* and therefore its CPU and RAM usage was not tested—although the total time does give an estimate.

The *Access Log* and *Error Log* fields represent the sizes of the respective logs—showing a 2 to 2.5 times increase in size. The size of the EBPF log (with-gretel only) was 5.374 ± 0.012 MiB. The size used by the gretel numbers in the error log was roughly 12.14 MiB, accounting for 40 % of the file (65 % of the increase). Similarly, gretel numbers accounted for 60 % of the access log size (97 % of the increase). A reduction in the size of each gretel number (e.g., from 256-bit to 64-bit) could, therefore, significantly reduce the size of the log file. More important, however, is the placement and quantity of logging points strongly influence log size—as seen by comparing the access log (fewer logging points) to the error log (additional logging points). Fewer logging points would bring the error log closer to the access log in size.

The memory usage at the end of the experiment, *RAM Anon*, represents the combined total amount of "anonymous" memory pages used by NGINX-A and NGINX-B. The amount of allocated memory at the start of the experiment followed the same proportions and was 0.044 MiB less in both configurations. The result shows that gretel numbers caused an 8 % increase in allocated memory. Such a minor increase aligns with expectations, since roughly four gretel numbers are stored per request, along with some temporary buffers for log and message generation. Parallel requests may have raised the memory requirements, but the results show that such an increase is unlikely to be significant (Nginx seems to preallocate most memory).

As for network traffic, each HTTP message (request and response) contains a single gretel number (and some text): resulting in exactly 74 additional bytes per message. In comparison, the header part of a typical (non-gretel) HTTP message ranges from 75 to 250 bytes. Each of the 10 thousand curl-invocation caused 4 HTTP messages (curl-to-A-to-B, B-to-A-to-curl), resulting in exactly 2.82 MiB additional network traffic. This corresponds to roughly 43 KiB per second, which is quite small, but this would accumulate with parallel requests and multiple applications. However, this seems unlikely to cause issues compared to the log sizes (which accumulate over time).

8. Discussion, limitations & future work

The POC workarounds, such as the use of EBPF instead of directly modifying the Linux kernel, were deemed sufficient to demonstrate the feasibility of gretel numbers. However, future work should evaluate the concept on a more realistic testbed (e.g., more applications beyond Nginx and logging an appropriate subset of system calls).

The proposed causality-tracking system aims to track *all* applications and *all* actions within a well-defined system (e.g., a website), but not necessarily *log* all actions. In essence: every action should be able to *account* for its cause when "asked". As demonstrated in the evaluation, the tracking itself is efficient, but the log space vs. accuracy trade-off

Table 2
Experiment results, comparing the performance impact with and without gretel numbers. The results show minimal CPU time and RAM overhead, but a significant increase in log size, when implementing gretel numbers.

Type	Duration	User Time	Sys. Time	Access Log	Error Log	RAM Anon
	(s)	**(s)**	**(s)**	**(MiB)**	**(MiB)**	**(MiB)**
without-gretel	64.5 ± 0.527	1.304 ± 0.034	3.937 ± 0.073	0.83 ± 0.0	13.842 ± 0.0	3.013 ± 0.004
with-gretel	66.7 ± 0.483	1.898 ± 0.103	5.576 ± 0.085	2.069 ± 0.0	32.448 ± 0.0	3.282 ± 0.006

must still be managed. Ultimately, the increased log size is the cost of cybersecurity, and this cost might be unavoidable.

The log size resulting from implementing gretel number tracking is influenced by two main factors: the bit size of each gretel number and the placement of logging points. Since gretel numbers dominate the log size, reducing their bit size (e.g., from 256-bit to 128-bit or 64-bit) could significantly reduce the log size. This would lead to a lower relative increase in log size (e.g., 30–75 %) compared to using 256-bit gretel numbers (a 150 % increase). However, the bit size must still be sufficient to maintain uniqueness. Similarly, the number of log entries is determined by the placement of logging points, which in turn has minimum requirements determined by (future) investigation questions and tamper-resistance. Hence, future work will explore bit size, forensic questions, and tamper-resistance.

Despite the advantages of gretel numbers, fuzzy correlation remains necessary in certain situations—owing to untracked external systems and limitations of tamper-resistance. Identifying such situations remains a topic for future work.

Similarly, while gretel numbers could be used for anomaly detection in IDS systems, this was not their original intended purpose. Future work should investigate whether gretel numbers could improve the accuracy of such algorithms.

9. Conclusion

This paper discussed how logging sufficiency in an IT system can be defined and potentially achieved—namely by anticipating questions asked during (hypothetical) future investigation. A key finding is that causal information is fundamental for answering many such questions, but is often missing from conventional logs. To address this issue, the paper proposes a novel logging system that uses event IDs, termed "gretel numbers", to track and record causality. A minimal prototype of the system was evaluated, showing the tracking to be highly efficient, but significantly increasing the necessary disk space. Space-saving measures will thus remain a topic for future work.

A logging system that inherently captures causal links could make log correlation easier, perhaps even trivial. In fact, much of the work done in forensic analysis, not just of logs, could be seen as reconstructing such causal links using limited data. Therefore, the study of causal links could contribute to the theory of digital forensics more broadly. This theory could be used to more accurately interpret digital evidence and maybe even define error rates (similarly to other forensic sciences). Furthermore, forensic datasets could model ground truth using causal links, leading to more precise validation of forensic tools (even if the tools themselves do not operate on gretel numbers), Lastly, the context provided by these links could potentially enhance anomaly detection and IDS algorithms.

Overall, this paper highlights logging design as an under-researched area in digital forensics. Even beyond logs, digital forensics is mostly concerned with artifacts that are useful by coincidence rather than by design. While this topic may be of less use in the day-to-day operations of Law Enforcement, IT organizations have the opportunity to proactively prepare their systems for future investigations by designing better evidence—and the specifics of such proactive practices warrant further study.

References

Azahari, A., Balzarotti, D., 2024. On the inadequacy of open-source application logs for digital forensics. Forensic Sci. Int.: Digit. Invest. 49, 301750. https://doi.org/10.1016/j.fsidi.2024.301750.

Barse, E., Jonsson, E., 2004. Extracting attack manifestations to determine log data requirements for intrusion detection. In: 20th Annual Computer Security Applications Conference, pp. 158–167. https://doi.org/10.1109/CSAC.2004.20.

Bhandary, M., Parmar, M., Ambawade, D., 2020. Securing logs of a system - an IoTA tangle use case. In: 2020 International Conference on Electronics and Sustainable Communication Systems. ICESC), pp. 697–702. https://doi.org/10.1109/ICESC48915.2020.9155563.

Bishop, M., 1990. A model of security monitoring. In: [1989 Proceedings] Fifth Annual Computer Security Applications Conference, IEEE Comput. Soc. Press, Tucson, AZ, USA, pp. 46–52. https://doi.org/10.1109/CSAC.1989.81024.

Daubner, L., Matulevičius, R., 2021. Risk-oriented design approach for forensic-ready software systems. In: Proceedings of the 16th International Conference on Availability, Reliability and Security. ACM, Vienna Austria, pp. 1–10. https://doi.org/10.1145/3465481.3470052.

Daubner, L., Buhnova, B., Pitner, T., 2024. Forensic experts' view of forensic-ready software systems: a qualitative study. J. Softw.: Evolut. Proc. 36 (5), e2598. https://doi.org/10.1002/smr.2598.

Gladyshev, P., Patel, A., 2004. Finite state machine approach to digital event reconstruction. Digit. Invest. 1 (2), 130–149. https://doi.org/10.1016/j.diin.2004.03.001.

Goel, A., Farhadi, K., Po, K., Feng, W.-c., 2008. Reconstructing system state for intrusion analysis. ACM SIGOPS - Oper. Syst. Rev. 42 (3), 21–28. https://doi.org/10.1145/1368506.1368511.

Gregg, B., 2013. Blazing Performance with Flame Graphs. URL. https://www.usenix.org/conference/lisa13/technical-sessions/plenary/gregg.

Hossain, M.N., Wang, J., Weisse, O., Sekar, R., Genkin, D., He, B., Stoller, S.D., Fang, G., Piessens, F., Downing, E., Chen, Y., Kim, T., Wenisch, T.F., Orso, A., Strackx, R., Lee, W., Zappala, D., Duan, H., Genkin, D., Yarom, Y., Hamburg, M., 2018. Dependence-Preserving Data Compaction for Scalable Forensic Analysis. USENIX, pp. 1723–1740.

International Organization for Standardization, 2013. Information Technology — Security Techniques — Information Security Management Systems — Requirements, Standard ISO 27001:2013. International Organization for Standardization, Geneva, CH. Oct.

Jacob, S., Qiao, Y., Lee, B., 2021. Detecting cyber security attacks against a microservices application using distributed tracing. In: Proceedings of the 7th International Conference on Information Systems Security and Privacy, pp. 588–595. https://doi.org/10.5220/0010308905880595.

Juma, N., Huang, X., Tripunitara, M., 2020. Forensic analysis in access control: foundations and a case-study from practice. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20. Association for Computing Machinery, New York, NY, USA, pp. 1533–1550. https://doi.org/10.1145/3372297.3417860.

Kenneally, E.E., 2004. Digital logs—proof matters. Digit. Invest. 1 (2), 94–101. https://doi.org/10.1016/j.diin.2004.01.006.

Kent, K., Souppaya, M., 2006. Guide to Computer Security Log Management, Tech. Rep. NIST Special Publication (SP) 800-92. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-92. Sep.

King, S.T., Chen, P.M., 2005. Backtracking intrusions. ACM Trans. Comput. Syst. 23 (1), 51–76. https://doi.org/10.1145/1047915.1047918.

Lee, K.H., Zhang, X., Xu, D., 2013. LogGC: garbage collecting audit log. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. CCS '13, Association for Computing Machinery, New York, NY, USA, pp. 1005–1016. https://doi.org/10.1145/2508859.2516731.

McClung, R.G., 1913. The Theory of Debit and Credit in Accounting. Mills and company, Boston, Morgan. Accessed: 2024-12-16). URL. http://archive.org/details/theoryofdebitcre00mcclrich.

Michael, N., Mink, J., Liu, J., Gaur, S., Hassan, W.U., Bates, A., 2020. On the forensic validity of approximated audit logs. In: Annual Computer Security Applications Conference. ACM, Austin USA, pp. 189–202. https://doi.org/10.1145/3427228.3427272.

Mitre Corporation, 2012. Mitre CEE Version 1.0-beta1. URL. https://cee.mitre.org/language/1.0-beta1/. (Accessed 2 December 2024).

OpenTelemetry Community, 2024. OpenTelemetry Specification 1.40.0, 2024-12-16, URL. https://opentelemetry.io/docs/specs/otel/.

Pangalos, G., Katos, V., 2010. Information assurance and forensic readiness. In: Sideridis, A.B., Patrikakis, C.Z. (Eds.), Next Generation Society. Technological and Legal Issues. Springer, Berlin, Heidelberg, pp. 181–188. https://doi.org/10.1007/978-3-642-11631-5_17.

Pasquale, L., Alrajeh, D., Peersman, C., Tun, T., Nuseibeh, B., Rashid, A., 2018. Towards forensic-ready software systems. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER '18. Association for Computing Machinery, New York, NY, USA, pp. 9–12. https://doi.org/10.1145/3183399.3183426.

Peisert, S., Bishop, M., Karin, S., Marzullo, K., 2007. Toward models for forensic analysis. In: Second International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE'07). IEEE, Bell Harbor, WA, USA, pp. 3–15. https://doi.org/10.1109/SADFE.2007.23.

Qiu, L., Song, X., Yang, J., Cui, B., 2022. Bee: end to end distributed tracing system for source code security analysis, highlights in science. Eng. Technol. 1, 209–218. https://doi.org/10.54097/hset.v1i.463.

Resnick, P., 2001. Internet Message Format, Request for Comments RFC 2822, Internet Engineering Task Force. Apr. https://doi.org/10.17487/RFC2822. URL. https://datatracker.ietf.org/doc/rfc2822. (Accessed 9 December 2024).

Rivera-Ortiz, F., Pasquale, L., 2020. Automated modelling of security incidents to represent logging requirements in software systems. In: Proceedings of the 15th International Conference on Availability, Reliability and Security, ARES '20. Association for Computing Machinery, New York, NY, USA, pp. 1–8. https://doi.org/10.1145/3407023.3407081.

Schneier, B., Kelsey, J., 1999. Secure audit logs to support computer forensics. ACM Trans. Inf. Syst. Secur. 2 (2), 159–176. https://doi.org/10.1145/317087.317089.

Shen, J., Zhang, H., Xiang, Y., Shi, X., Li, X., Shen, Y., Zhang, Z., Wu, Y., Yin, X., Wang, J., Xu, M., Li, Y., Yin, J., Song, J., Li, Z., Nie, R., 2023. Network-centric distributed tracing with DeepFlow: troubleshooting your microservices in zero code. In: Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23. Association for Computing Machinery, New York, NY, USA, pp. 420–437. https://doi.org/10.1145/3603269.3604823.

W3C, 2021. Trace Context. URL. https://www.w3.org/TR/trace-context/. (Accessed 10 October 2024).

Wang, W., Daniels, T., 2005. Building evidence graphs for network forensics analysis. In: 21st Annual Computer Security Applications Conference. ACSAC'05), pp. 11–266. https://doi.org/10.1109/CSAC.2005.14.