Efficient Fingerprint Matching for Forensic Event Reconstruction

Tobias Latzo(✉)

Department of Computer Science,
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU),
Erlangen, Germany
tobias.latzo@fau.de

Abstract. Forensic investigations usually utilize log files to reconstruct previous events on computing systems. Using standard log files as well as traces of system calls, we analyze what traces are left by different events on a GNU/Linux server that runs different common services like an SSH server, Wordpress, Nextcloud and Docker containers. Based on these traces, we calculate characteristic fingerprints of these events that can later be matched to other log files to detect them. We develop a matching algorithm and examine the different parameters that influence its performance both in terms of event detectability and detection time. We also examine the effect of using different subsets of system calls to improve matching efficiency.

Keywords: Forensic event reconstruction · Linux logs · System call tracing · SIEM

1 Introduction

One of the main tasks of forensic analysts is to reconstruct what happened on a system in the past. To do so, they examine traces that are generated by actions or events that were executed on the system. One of the most prominent example of such traces are log messages in system or application log files [7]. Well-known examples in GNU/Linux systems are the `syslog` file and the file `auth.log`. On Windows, the central logging mechanism is the Windows Event Log [18]. Typically, these log files may contain a lot of log data giving insight into many different events from the past. However, there are also many events that do not appear in the common system logs, e.g., removing a file by a user with the proper access rights, connecting to a remote server, or shredding a file [14].

Security Information and Event Management (SIEM) systems aim at collecting (security relevant) information from log files, extract information from them and correlate this information with data from different computer systems with the goal to detect security relevant activities like incidents. However, these systems also struggle with imprecise and incomplete information if they depend

S. Goel et al. (Eds.): ICDF2C 2020, LNICST 351, pp. 98–120, 2021.
https://doi.org/10.1007/978-3-030-68734-2_6

on log files alone [17] and so other sources of information are used as well. One such source are the sequence of system calls that are executed on a particular system. It is well-known that system call traces help in detecting malicious and benign system activities [14,20]. However, it is also well-known that in terms of performance it is unrealistic to trace *all* system calls in a computing system. The resulting research question that underlies this paper is how system call traces can be used together with common system logs in a meaningful and efficient way to enable forensic event reconstruction?

1.1 Related Work

There is a lot of research in the field of incident detection based on log files. Most often these works focus on definitely malicious events. Examples of this stream of work are UCLog [15] (Unified Correlated LOGging architecture for intrusion detection) and its successor UCLog+ [25]. These tools correlate logging information from different sources like kernel API loggers (i.e., system calls), network loggers, file system loggers, etc., and generate an alarm in a suspected case. There also many approaches that utilize system call sequences to classify malware [12,13].

Another approach called "computer profiling" [16] infers events from cause-effect rules that are specified by an examiner. Thus, this approach is only partially automatable. Also similar is the approach of Kahn et al. [11] who make use of a neural-network to learn signatures based on file system metadata. Since neural networks need a huge amount of training data, this approach does not scale, too. Other research uses hidden Markov models [19,24]. Furthermore, there is an ontology-based approach for Windows systems [4] that generates timelines based on log files.

Gladyshev and Patel [9] pioneered the areas of formalizing event reconstruction using finite state machine representations of computing systems and applying formal logic to detect events [8]. Based on this research, there exists a slightly more practical approach that computes signatures of system events [10]. The resulting signatures are quite complex and probabilistic and were not applied to the reconstruction based on log files.

In this paper, we make use of the theory of forensic event reconstruction by Dewald [5]. This work was used by Latzo and Freiling [14] who examined the value of logging information for forensic event reconstruction. The work reveals that many events do not leave any traces in system log files. For enhancing event reconstruction, they utilized system call traces as an additional log source. The size of characteristic fingerprints (see also Sect. 2.2) is regarded as a quality criteria for forensic fingerprints. These sizes are examined for different feature sets and log sources. It turns out that using system call traces as log data is very beneficial in terms of the quality of characteristic fingerprints. It is obvious that tracing all system calls is expensive in terms of performance and the corresponding system cannot be used as a productive system anymore. In this paper, we want to calculate the characteristic fingerprints with a bigger event set, analyze how matching performs with that approach and show how to make matching more efficient.

1.2 Contribution

In this paper, we systematically analyze what and where traces are left by different events and how to utilize them for forensic event reconstruction by calculating characteristic fingerprints. We further study how these characteristic fingerprints perform in matching those events. We also utilize system call traces as an additional log source. Since tracing all system calls is extremely expensive in terms of performance, we show how to systematically reduce tracing overhead but still being able to detect events.

We make use of Dewald's theory of forensic event reconstruction [5] that was also applied by Latzo and Freiling [14] to log files. We build upon their research and calculate characteristic fingerprints on a larger event set. We additionally evaluate matching performance and the role of system calls in forensic event reconstruction and how to do that efficiently.

The main insights of our experiments are as follows:

1. Basically, we are able to reproduce the work of Latzo and Freiling [14] and show that our characteristic fingerprints are quite similar. Furthermore, we extended their event set by 30 events.
2. We show that characteristic fingerprints are actually applicable for forensic event reconstruction. The matching results are quite good. Overall, the sensitivity for our complete event set is about 88% and reaches 100% if only detectable events are considered.
3. We show how to systematically use system calls for forensic event reconstruction. Furthermore, we reveal what system calls are discriminative and how expensive these are in terms of performance.

1.3 Outline

The paper is structured as follows: First, in Sect. 2 we give some background information on system call tracing and the theory of forensic fingerprint calculation. In Sect. 3 we give insights into the experimental setup we used for our measurements. Further in Sect. 4 we calculate (characteristic) fingerprints for a large event set. Performance of the matching is analyzed in Sect. 5. Eventually, in Sect. 6 we investigate the role of system calls for forensic event reconstruction and how to use them efficiently.

2 Background

2.1 System Call Tracing

Operating systems offer an interface—the *system calls*—for user applications (and libraries) to perform specific actions usually on shared resources. Examples for GNU/Linux are: `open`, `read` or `write` a file or `execve` to spawn a new process, etc. Basically all interaction with a user or the machine, i.e., writing to a terminal or file, displaying something on the screen, sending a network package, etc. can only be performed using system calls.

Hence, system call traces disclose a lot about the behavior of a program. And so system call traces are a huge research topic and there are multiple ways to get those. One traditional approach is to run an application in a sandbox [23] which is used for dynamic analysis. In this approach, only system calls of a process (can have multiple threads) are monitored, though. Using virtual machine introspection allows to monitor all system calls on a system. This means that one does not need to know in advance what process or thread shall be monitored. One example is `libvmtrace` [21] that we used for our measurements. This software also allows to trace only a defined set of system calls which can be very beneficial in terms of performance. Basically, tracing *all* system calls cannot be used in practice for performance reasons.

2.2 Forensic Fingerprint Calculation

User actions—henceforth called *events*—on a computer system leave specific patterns. These patterns can be used as *forensic fingerprints* as they can help an analyst to determine if an event happened on a system, or not. In this paper, we make use of Dewald's [5] definitions of forensic fingerprints. In the following we want to explain briefly the adaption of this definitions for log messages.

Basically, a log message can be described by a set of features, i.e., a *feature vector*. In the following there are a few examples of possible features:

- source: the source, form where the log message comes from
- type_id: describes the kind of log message (e.g., a user login),
- time: a time stamp of the event,
- user: the login name (i.e., who is affected?),
- path: a path that is associated with the event,
- etc.

There are many features that are not very useful for fingerprint generation, e.g., the time stamp or the user. Otherwise, it would only be possible to detect an event of a specific user at a specific time. However, there is a set of *relevant features* F. Finally, V is defined as the set of all possible feature vectors over F.

An event that occurs may trigger entries in log files. Thus, an event can also be regarded as a generator of log files, i.e., of feature vectors. Σ is defined as a set of all events that may happen in a computer system. A single event $\sigma \in \Sigma$ generates a set of feature vectors. This feature vectors generated by σ can be regarded as the trace left by σ. The *Evidence Set* $E(\sigma)$ [5] is defined as the set of all subsets of feature vectors in V generated by σ. Partial evidence is also evidence, so formally $E(\sigma)$ must be closed under subsets (which is also a technical requirement for calculations).

Overall, $E(\sigma)$ contains *all* feature vectors that σ generates. So it is clear, that evidence sets of different events are overlapping. Especially during a forensic analysis, it is important to determine whether an event σ happened or did not happen, though. This means, one needs to know which feature vectors are caused by σ and not by any other event. For this reason, *characteristic fingerprints* are

calculated. The characteristic fingerprint $CE(\sigma)$ of $\sigma \in \Sigma$ is defined with respect to a *reference set* of other events $\Sigma' \subseteq \Sigma$:

$$\mathrm{CE}(\sigma, \Sigma') = E(\sigma) \setminus \bigcup_{\sigma' \in \Sigma'} E(\sigma')$$

The characteristic fingerprint $\mathrm{CE}(\sigma, \Sigma')$ can be regarded as the set of feature vectors of the evidence set $E(\sigma)$ that remains when one subtracts all feature vectors that are also part in any other evidence set $E(\sigma')$ with $\sigma' \in \Sigma'$. For forensically-sound evidence it is necessary that $|\mathrm{CE}(\sigma, \Sigma')|$ (henceforth also called *size*) and $|\Sigma'|$ are sufficiently large. Then one can assume that σ happened and not any other event of the reference set $\sigma' \in \Sigma$. $\mathrm{CE}(\sigma, \Sigma')$ may also have not have any feature vector. This means, σ cannot be detected reliably with the given feature vectors F. In this paper, the sizes of a bigger event set are investigated. Furthermore, this paper gives researches the matchability of characteristic fingerprints and how to make it efficient.

3 Experimental Setup

We now briefly describe the setup and architecture that was used for the measurements in this paper. In principle, we use a similar setup as Latzo and Freiling [3] while our software also comes with a Matching Engine.

3.1 Scenario and Attacker Model

Our setup consists of a GNU/Linux server running Ubuntu 16.04 with several services enabled: a Wordpress instance, an SSH-server, Docker containers and a Nextcloud instance. We assume that is may be a typical infrastructure in small and medium-sized enterprises. There such servers are a valuable asset and might be interesting for attacks as well as forensic analysis.

We assume a high-privileged attacker with root privileges that he or she might have gained by being either being the system administrator or via a privilege escalation attack. This makes it easy for the attacker to manipulate log messages and cover his or her traces after compromising.

3.2 Architecture and Implementation

In Fig. 1 one can see a simplified schema of the architecture of our experimental setup. The target system (Ubuntu 16.04) runs virtualized as a Xen guest on an x86-64 CPU and two GiB of RAM. System log files are drained via Filebeat [6]. A monitor VM using `libvmtrace` [21] instruments the target system and traces all requested system calls. Note, tracing of system calls happens on another virtual machine, so basically it is not possible for an attacker with root privileges on the target system to disable system call tracing. By default, we trace *all* system calls. Note, this highly impacts performance which is treated later. The system

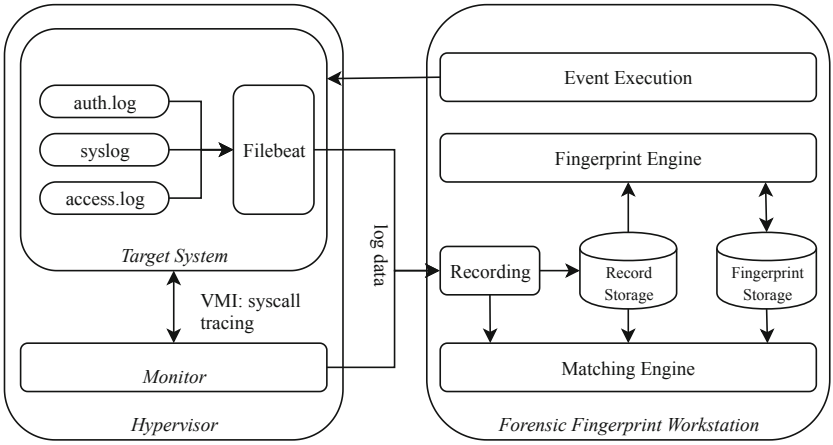

Fig. 1. Architecture of our experimental setup.

log files and the system calls are sent via Apache's Kafka [22] to the Forensic Fingerprint Workstation.

To calculate a new fingerprint for an event σ, the event is executed automatically multiple times from the Forensic Fingerprint Workstation. Most events can be executed via an open SSH session. The Recording module receives all logs (including system call traces) via Kafka and stores them in a unified log format as JSON files. To get the data in a uniform format, the log messages are parsed using regular expressions into the uniform log format—a feature vectore. Afterwards, the corresponding fingerprint can be calculated by the Fingerprint Engine as described in Sect. 2.2.

After calculating multiple fingerprints (in our case all), one can calculate characteristic fingerprints (also described in Sect. 2.2). All resulting characteristic fingerprints are also stored as JSON files.

There are two ways to start matching. First, it is possible to perform live matching. This means that all incoming logs (and system call traces) are processed directly by the Matching Engine. We assume that the feature vectors of any event all happen within at most Δ time units. For each enabled (characteristic) fingerprint the Matching Engine checks how many feature vectors of the fingerprint do match with the incoming log vectors within a time of 2Δ (in order to not miss an event that occurs between two such time slots). The second possibility is to match on recorded logs. In this case, the Matching Engine calculates the percentage of equal feature vectors in the characteristic fingerprint and the stored log. This is the way we use the Matching Engine in this paper and allows reproducibility.

In our evaluation, we only use a single target system. Log entries only need to be sent via Kafka. Thus, it is basically also possible to have multiple target systems.

4 Characteristic Fingerprints of a Large Event Set

In this section, we want to calculate characteristic fingerprints for a big event set, i.e., we calculate characteristic fingerprints similar as Latzo and Freiling [14] but for 45 events.

4.1 Event Set

To make the results comparable with the measurements of Latzo and Freiling [14], we extended the event set they use with "similar" events. This means, we also evaluate events that are usually performed by server administrators via the Linux command line. Since we want to analyze the genericity of the approach of forensic fingerprint generation and matching, it is actually not important what events are used and what an event does, i.e., if an event itself is malicious or not. In general, it is not possible to state whether an administrator's user input is malicious or not. Basically, all events can be also used in a malicious context, be it information retrieval, removal of traces, etc. Thus, we assume all events are potentially interesting for a forensic analyst.

All events are executed (and therefore recorded) 40 times (training set). Then for each event a fingerprint is calculated. For this purpose, we used the threshold of 0.8, i.e., features had to occur in the data of 80% of the 40 runs. The work of Latzo and Freiling [14] revealed that the more features are in F, the bigger the corresponding characteristic fingerprints. A possible drawback of using *too* many features for the (characteristic) fingerprints is that the feature vectors may become too specific and only work in a certain context, e.g., with a dedicated user. We decided to use the maximum feature set of [14]:

- `source`: from what file does the log come from,
- `type_id`: the type of the log message,
- `path`: a path that is related with the log message and
- `misc`: a miscellaneous field that may contain different information, e.g., a network adapter.

We assume that in our events, this feature set should not be too specific. All calculations of (characteristic) fingerprints are performed using this feature set. Table 2 shows the total amount of feature vectors in the fingerprints by origin. It is striking that system calls are clearly dominating the feature vectors in a fingerprint. Furthermore, depending on the class, an event typically leaves traces in other logs [14]. Note, the way we record events is rather error-prone. Tracing all system calls makes it hard to determine a good Δ since the actual duration of an event may vary. Especially for common log messages, it is possible that these do not become part of a fingerprint, when the variation is so big that they do not occur in 80% of the rounds.

Figure 2 gives an impression of the similarity of event fingerprints. The heatmap indicates what percentage of Fingerprint F_1 is overlapped by F_2. An "X" stands for a full overlap. Events with a larger Δ usually other events more

Table 1. List of events that is used for the evaluation. The corresponding event is recorded for Δ seconds.

Class	Name	Description	Δt
CLI	ls	Lists files	5
	cp	Copies file	5
	mv	Moves file	5
	cat	Cats file	5
	vmstat	Virtual memory statistics	5
	netstat	Network statistics	5
	tar	Creates compressed tar archive	5
	rm	Removes file	5
	shred	Shreds file	5
	curl	Downloads file	5
CLI root	tailShadow	Reads /etc/shadow	10
	vimHosts	Opens /etc/hosts in Vim	10
	rmSudo	Removes file with sudo	10
	shredSudo	Shreds file with sudo	10
Web	wordpressLogin	Wordpress Login	20
	wordpressSearch	Wordpress Search	20
	wordpressOpen	Opens Wordpress website	20
Service	sshLogin	SSH login (server side)	30
	apacheStop	Stops apache web server	110
	mysqlWp	Login into Wordpress DB via command line	20
Kernel modules	lsmod	Lists loaded kernel modules	10
	insmod	Loads kernel module	5
	rmmod	Unloads kernel module	5
Docker	dockerHelloWorld	Starts docker hello world example	105
	dockerUbuntuLog	Starts docker ubuntu and show log	110
	dockerImages	Lists all docker images	10
	dockerPs	Lists all running dockers	10
	dockerPSA	Lists all dockers container	10
	dockerUbuntuSleep	Starts docker in background	100
	dockerRm	Removes all docker containers	15
	dockerNginx	Runs nginx docker and curl it	80
	dockerUbuntuBash	Attaches bash of container	15
	dockerPrune	Removes unused container	60
	dockerPruneVolumes	Removes unused objects and volumes	60
	dockerRmImages	Removes all images	60
	dockerUbuntuBashCp	Attaches container and runs cp	95
	dockerUbuntuBashMv	Attaches container and runs mv	95
	dockerUbuntuBashRm	Attaches container and runs rm	95
	dockerUbuntuBashCat	Attaches container and runs cat	95
Nextcloud	nextcloudStatus	Shows Nextcloud status	35
	nextcloudAppList	Lists Nextcloud apps	40
	nextcloudUserList	Lists Nextcloud user	40
	nextcloudUserAdd	Adds new Nextcloud user	65
	nextcloudGroupList	List Nextcloud groups	40

than events with small Δ. Fingerprints usually contain a lot of feature vectors that are related to other events that are running simultaneously in background. We treat those feature vectors as *noise*. Usually, most overlapping feature vectors are related to noise. Fingerprints of events with very small Δ, e.g., events from the class CLI are overlapped with rather bigger proportions. It is also striking that fingerprints within a class do have higher overlapping rates, e.g., some Docker fingerprints even fully overlap some other docker fingerprints.

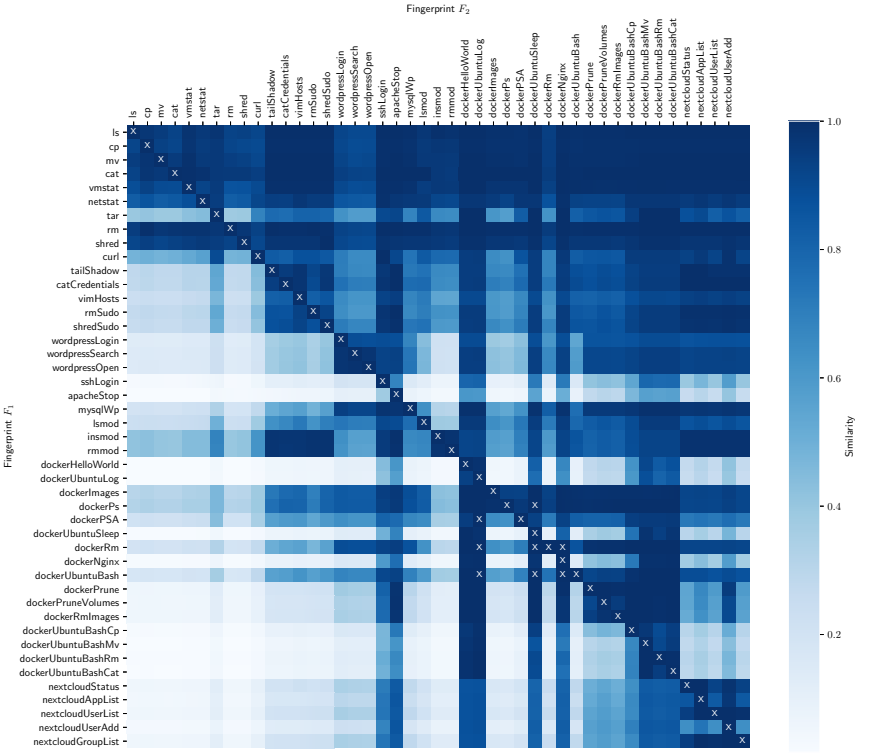

Fig. 2. The heatmap shows what percentage of F_1 is overlapped by F_2. An "X" indicates that F_2 fully overlaps F_1.

4.2 Results

Table 3 shows the number of feature vectors of characteristic fingerprints by their origin. The reference set (see also Sect. 2.2) is always the set of all other events.

It is noteworthy that the system calls clearly dominate the characteristic fingerprints. While there are several entries in the corresponding fingerprints from other log sources (see also Table 2), these vectors are no longer part of the characteristic fingerprints. If one compares the results with Latzo and Freiling [14], one can see that the sizes of characteristic fingerprints are in the same order

of magnitude than ours while our event set is about three times as big. So, the characteristic fingerprints are considerably more *stable* regarding our reference set.

There are also seven zero entries. This means that for these event, it was not possible to calculate a characteristic fingerprint with all other entries in the reference set. Since now the event set is bigger, it is more likely that some events are fully overlapped by other events (see also Fig. 2).

While Latzo and Freiling [14] observed a relation between the size of an event (with respect to Δ) and the size of the corresponding characteristic fingerprint, with our larger event set we can not fully confirm this. Docker events are quite large and also do have rather large fingerprints. However, the size of the characteristic fingerprints is not in the same order of magnitude as for example the service events. So we assume that the duration of the event has not such a big impact as expected.

The last column of Table 3 contains the costs of the characteristic fingerprints. How these costs are calculated is shown later in Sect. 6.

5 Matching

We could show that it is possible to calculate characteristic fingerprints for most events for even a bigger reference set Σ'. In this section we want to analyze how the characteristic fingerprints perform in terms of event reconstruction by matching on event traces.

5.1 Methodology

To evaluate matching with characteristic fingerprints, we executed each event ten times (test set) and saved the corresponding traces as described in Sect. 3. Then, the Matching Engine calculates for all traces a score. Let $T(\sigma')$ be the trace of the event σ', then the score is calculated as follows:

$$\text{score}(\text{CE}(\sigma, \Sigma'), T(\sigma')) = \frac{|\text{CE}(\sigma, \Sigma') \cap T(\sigma')|}{|\text{CE}(\sigma, \Sigma')|}$$

This means that the score of a characteristic fingerprint $\text{CE}(\sigma, \Sigma')$ is the proportion of matched feature vectors of $T(\sigma')$ in the characteristic fingerprint. The term *matched* means that all relevant Features F that were used for calculating the fingerprint are the same in the trace vector and in the fingerprint vector. Ideally, $\text{score}(\text{CE}(\sigma, \Sigma'), T(\sigma)) = 1$ while $\text{score}(\text{CE}(\sigma, \Sigma'), T(\sigma')) = 0$ for $\sigma' \neq \sigma$.

5.2 Matching Results

Figure 3 shows the results of our matching experiment. The heatmap shows the average scores (formula is shown above for one execution) for ten executions of the event. The higher the average matching score, the darker the cell. Basically,

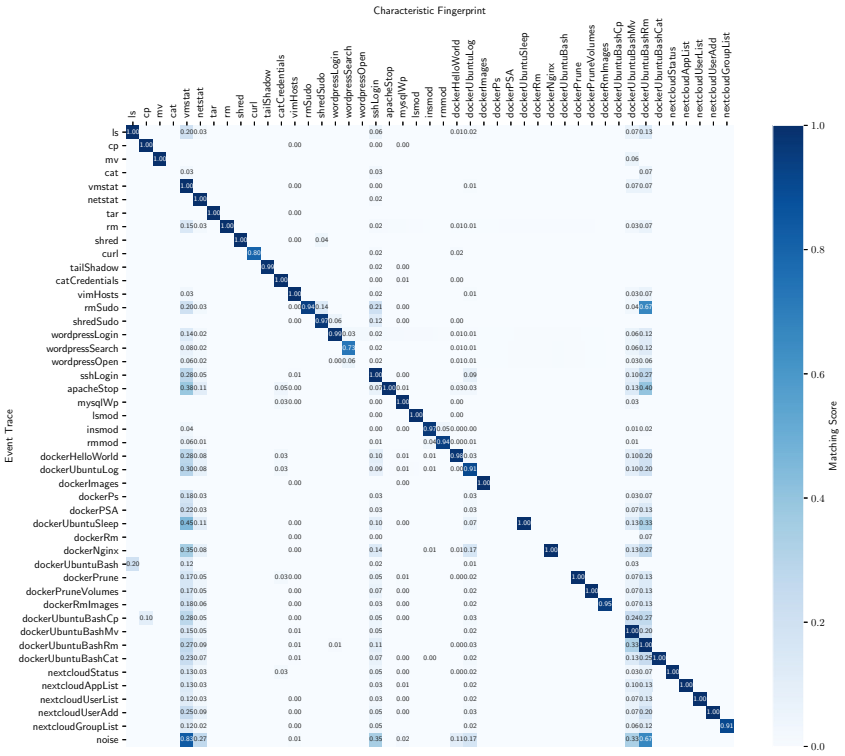

Fig. 3. The heatmap shows a matching matrix before subtracting noise from the characteristic fingerprints. On the y-axis, the ground truth, i.e., the actual event traces are listed while on the x-axis are the corresponding characteristic fingerprints. The average matching scores of ten traces is represented by the darker coloring and average score.

the matching results in Fig. 3 look considerably decent. For all existing characteristic fingerprints, the average matching score is rather high while overall the matching scores of the false events are almost everywhere below 0.1. The last line in the matrix shows the matching of one hour of noise, i.e., in this hour no user interaction with the system happened but only standard background tasks are running. There, the matching scores are also very low with one exception (`vmstat`). Optimally, the matrix would only have a one-diagonal while the rest are zero entries. Since we use *characteristic* fingerprints for matching, a similar graph is expected.

Figure 3 also shows that only some characteristic fingerprints produce false matches (even if the score is usually quite low). However, these characteristic fingerprints usually also do match noise (also with usually rather low scores). By researching the false matches, it is noticeable that matching feature vectors in these cases come from still present noise in the corresponding characteristic fingerprints. For example, there are some system calls related to the Filebeat [6]

(the software we use to drain log files) and to Xymon [1] (a system monitoring software).

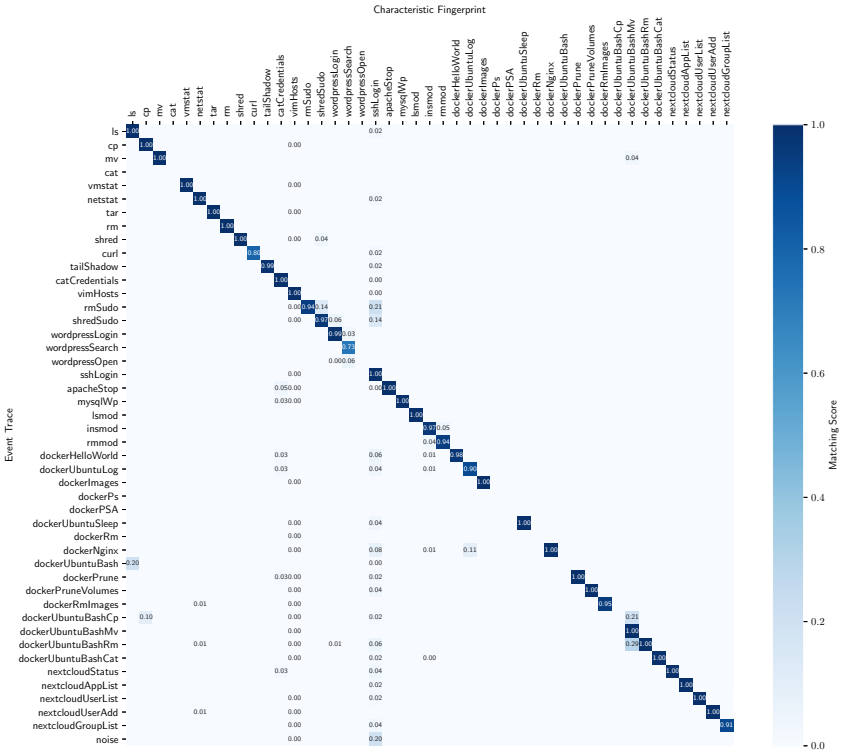

Fig. 4. The heatmap shows a matching matrix after subtracting noise from the characteristic fingerprints. On the y-axis, the ground truth, i.e., the actual event traces are listed while on the x-axis are the corresponding characteristic fingerprints. The average matching scores of ten traces is represented by the darker coloring and average score.

To get rid of the matching noise in the characteristic fingerprints, we subtract the noise from the characteristic fingerprints and perform matching again. The results can be seen in Fig. 4. There are much fewer false matching entries and the corresponding matching scores are also much lower. Thus, subtracting noise from characteristic fingerprints has turned out to be beneficial. Only `vimHots` and `sshLogin` still seem to contain some noise.

Figure 5 shows the corresponding *Receiver Operating Characteristic curves* (ROC). The ROC curves compare the true positive rate—also called *sensitivity*—with the false positive rate while varying the matching threshold. This value says above which threshold a matching score is interpreted as a match. Especially, using Fig. 4, it is easily possible to determine a perfect threshold (sensitivity of 1) and false positive rate of 0. An example of such a threshold would be 0.7.

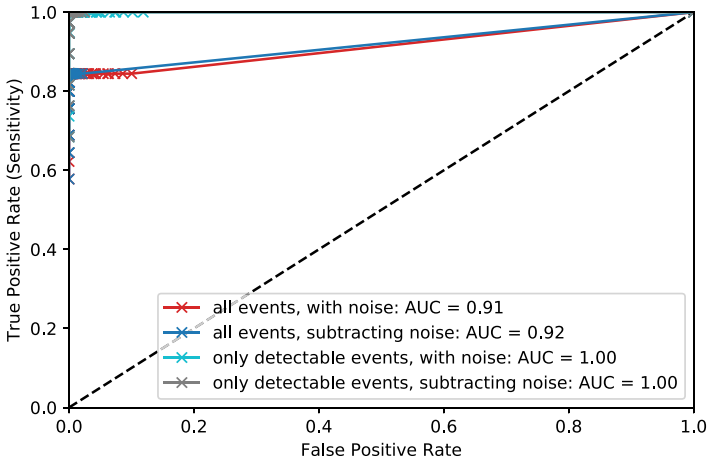

Fig. 5. ROC curves of matching with and without noise and for all events and only detectable events.

In Fig. 3, one can see that using this threshold would produce a false positive. However, this single false positive has such a small impact on the false positive rate that it is not really visible in the corresponding ROC curves. Figure 5 also shows the ROC curves when only considering detectable events. Since we can then find a perfect threshold (when subtracting noise), the corresponding *Area Under the Curve* (AUC) is 1.0.

6 System Calls for Forensic Event Reconstruction

The results in the previous sections show that system calls are dominating characteristic fingerprints so also play an important role for forensic event reconstruction. Many events could not be detected using common system log sources. The Δ values in Table 1 indicate that tracing all system calls makes the system nearly unusable, though. In this section, we want to give an overview of the distribution of system calls occuring on a system. Based on this distribution, we define a cost function for system calls that allows to make a statement about the *costs* of a characteristic fingerprint. Furthermore, this section shows what set of system calls is discriminatory, i.e., is useful for forensic event reconstruction.

6.1 System Call Distribution in System Activity and in Characteristic Fingerprints

To get the distribution of system calls occurrences, we record one hour of noise in the system. So, all system calls are traced for an hour while no user interaction is happening. Then, all records of the events (40 rounds per event) and the

noise recording are merged together to get a representative list of system calls occurring on the system A. This list contains more than 41 million system calls.

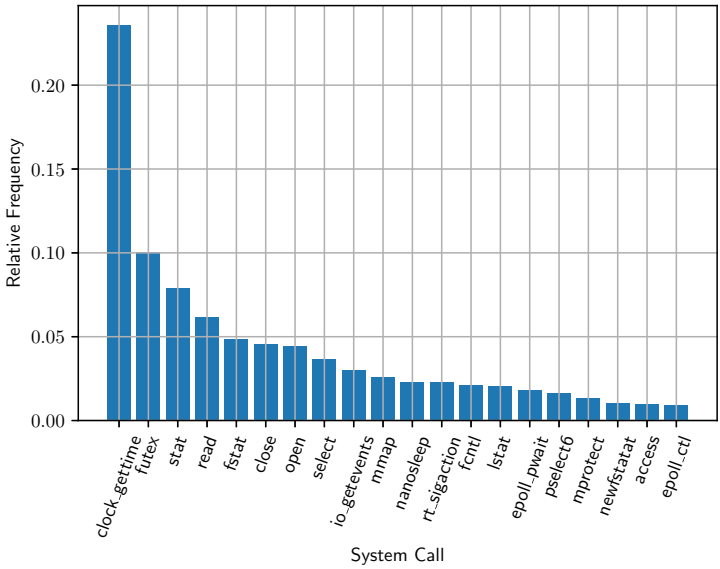

Fig. 6. Typical distribution of system calls in a system (using reference set A).

Figure 6 shows a histogram of the 20 most frequent system calls in our set of system calls. The most frequent system call (24%) is `clock_gettime` which—as the name suggests—is responsible for returning the current time. `futex` is used for synchronisation while the next next system calls `read`, `fstat`, `close` and `open` are all related to basic file operations. Since Linux is a file-based system, it is no surprise that these kind of system calls are quite frequent.

The histogram in Fig. 7 shows the absolute numbers of system calls in characteristic fingerprints. Here, we also see that file-related system calls like `open` and `stat` are also important in characteristic fingerprints. Table 4 lists these system calls and describes their purpose in more detail. Basically, one can say that file-related system calls are quite generic and occur in many characteristic fingerprints. Other system calls in this list are used more for special purpose like `finit_module` and `delete_module` that are used for loading and unloading Linux kernel modules and so clearly belong to the corresponding events (see also Table 1).

In the following section, we want to show how to make the system call tracing more efficient by tracing fewer system calls based on the knowledge of the distributions of system calls in the system and in characteristic fingerprints.

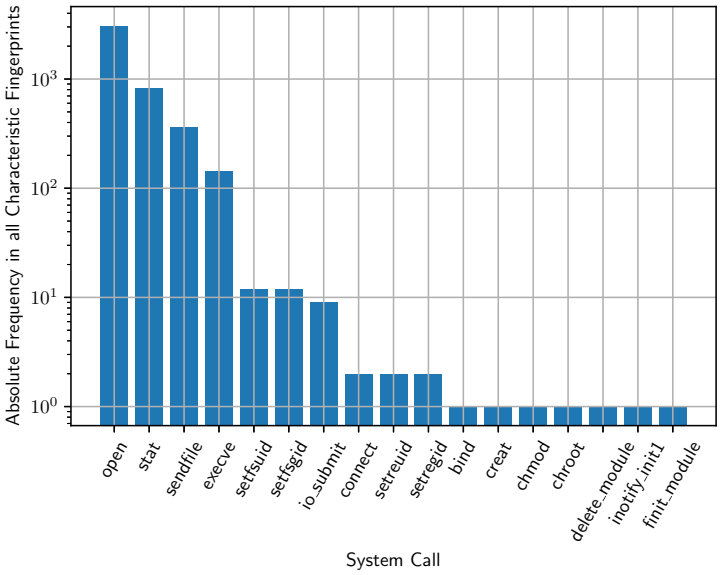

Fig. 7. Absolute occurences of system calls in characteristic fingerprints.

6.2 The Cost Function

In this section we want to develop a cost function for characteristic fingerprints. This function shall be used to assess how expensive it is to trace all system calls that are used in a characteristic fingerprint.

Basically, `libvmtrace` [21] (the software we use for system call tracing) allows to trace a defined set of system calls. For example, only for a specific set of system calls, there is a trap into the hypervisor. Traps into the hypervisor are quite expensive and so if one traces a lot of system calls, there are a lot of traps and context switches which strongly decreases the performance. We assume every trace of a system call has the same constant costs as the most expensive part is trapping into the hypervisor that is the same for every system call.

Let s be a system call that should be traced and A be a representative list of system calls that happen over a long period of time in the system. Then we define the costs c of s as the relative frequency of the system call in the representative system activity A, i.e., the number of times that s occurs in A over the total length of the list. Overall, we traced about 320 different system calls.

Table 4 lists the system calls that appear in characteristic fingerprints, describes their purpose and shows the (rounded) costs of the system calls that appear in characteristic fingerprints. It is striking that most system calls that are part of characteristic fingerprints should not have an impact on performance. The two most frequent system calls in characteristic fingerprints (`open` and `stat`) are the most expensive in this table, though. However, the table shows that tracing most system calls is not necessary.

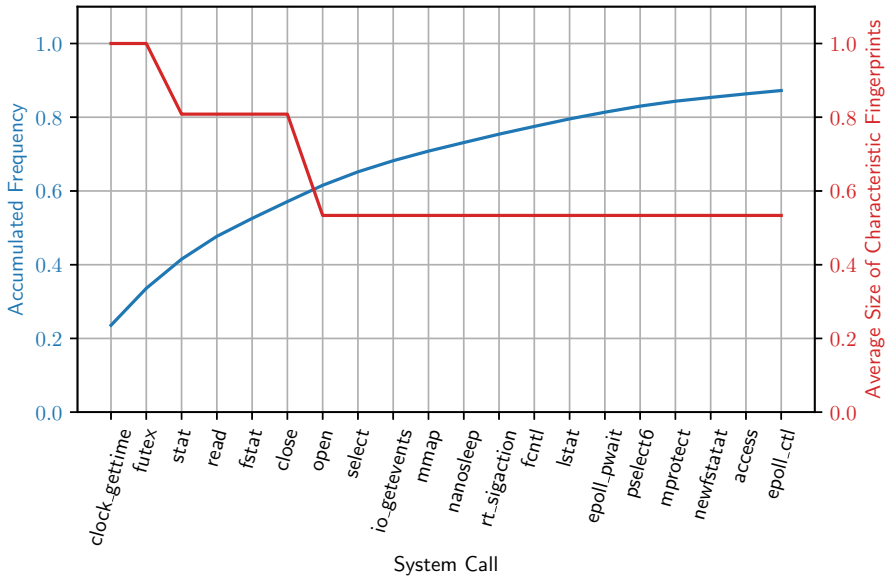

Fig. 8. The accumulated frequency of system calls occurrences (in blue) and the factor of reduction of the size of characteristic fingerprints when successively tracing fewer system calls. (Color figure online)

Based on the formula above, we can define a cost function for characteristic fingerprints. Let $\mathrm{CE}(\sigma, \Sigma')$ be a characteristic fingerprint, then the cost of a characteristic fingerprint is defined as follows:

$$c(\mathrm{CE}(\sigma, \Sigma')) = \sum_{s \in \mathrm{CE}(\sigma, \Sigma')} c(s)$$

So, the costs of a characteristic fingerprint $\mathrm{CE}(\sigma, \Sigma')$ is the sum of the relative frequencies of system calls that are part of the characteristic fingerprint. In the last column of Table 3 are the cost of each characteristic fingerprint when tracing all possible system calls. It shows that the maximum costs of a characteristic fingerprint is only about 0.125. Nearly 88% of all system calls do not need to be traced which would increase performance a lot. However, 12.5% of the performance overhead is a lot (see also Table 1). The following shows how to reduce further the costs of characteristic fingerprints and the impact on the size of the characteristic fingerprints.

6.3 Greedy Elimination of Expensive System Calls

About 88% of occurring system calls do not need to be traced since they are not part of any characteristic fingerprint, anyway. Obviously this should increase performance already a lot. However, even 12.5% of the performance overhead is

Table 2. Sizes of fingerprint sets classified by origin.

Class	Name	Source				Total
		syslog	auth.log	access.log	syscalls	
CLI	ls	0	0	0	1386	1386
	cp	0	0	0	1401	1401
	mv	0	0	0	1382	1382
	cat	0	0	0	1315	1315
	vmstat	0	0	0	1457	1457
	netstat	0	0	0	1614	1614
	tar	0	0	0	3367	3367
	rm	0	0	0	1308	1308
	shred	0	0	0	1372	1372
	curl	0	0	0	3173	3173
CLI Root	tailShadow	0	4	0	2944	2948
	catCredentials	0	4	0	3032	3036
	vimHosts	0	1	0	3100	3101
	rmSudo	0	1	0	2932	2933
	shredSudo	0	1	0	2649	2650
Web	wordpressLogin	0	0	19	9596	9615
	wordpressSearch	0	0	10	8250	8260
	wordpressOpen	0	0	10	8096	8106
Service	sshLogin	1	4	0	32380	32385
	apacheStop	0	4	0	45984	45988
	mysqlWp	0	0	0	4644	4644
Kernel Modules	lsmod	0	0	0	3776	3776
	insmod	1	2	0	3485	3488
	rmmod	1	1	0	3474	3476
Docker	dockerHelloWorld	7	0	0	60928	60935
	dockerUbuntuLog	0	0	0	57208	57208
	dockerImages	0	0	0	3345	3345
	dockerPs	0	0	0	5107	5107
	dockerPSA	0	0	0	8057	8057
	dockerUbuntuSleep	4	0	0	61405	61409
	dockerRm	0	0	0	5992	5992
	dockerNginx	0	0	0	44231	44231
	dockerUbuntuBash	0	0	0	5051	5051
	dockerPrune	0	0	0	21802	21802
	dockerPruneVolumes	0	0	0	21622	21622
	dockerRmImages	0	0	0	4194	4194
	dockerUbuntuBashCp	4	0	0	73529	73533
	dockerUbuntuBashMv	4	0	0	53373	53377
	dockerUbuntuBashRm	4	0	0	57698	57702
	dockerUbuntuBashCat	4	0	0	73032	73036
Nextcloud	nextcloudStatus	0	4	0	24768	24772
	nextcloudAppList	0	4	0	26830	26834
	nextcloudUserList	0	4	0	24936	24940
	nextcloudUserAdd	0	4	0	30966	30970
	nextcloudGroupList	0	4	0	21764	21768

a lot (see also Table 1). In the following we show how to reduce further the costs of characteristic fingerprints and the impact on the size of the characteristic fingerprints.

Table 3. Sizes and costs of characteristic fingerprints.

Class	Name	Source				Total	$c(\mathrm{CE}(\sigma, \Sigma'))$
		syslog	auth.log	access.log	syscalls		
CLI	ls	0	0	0	1	1	0.001
	cp	0	0	0	4	4	0.124
	mv	0	0	0	2	2	0.08
	cat	0	0	0	0	0	0
	vmstat	0	0	0	6	6	0.045
	netstat	0	0	0	15	15	0.045
	tar	0	0	0	5	5	0.08
	rm	0	0	0	1	1	0.001
	shred	0	0	0	2	2	0.045
	curl	0	0	0	1	1	0.044
CLI Root	tailShadow	0	1	0	6	7	0.08
	catCredentials	0	1	0	3	4	0.045
	vimHosts	0	1	0	219	220	0.124
	rmSudo	0	0	0	2	2	0.001
	shredSudo	0	2	0	7	9	0.124
Web	wordpressLogin	0	0	7	56	63	0.123
	wordpressSearch	0	0	1	2	3	0.123
	wordpressOpen	0	0	0	0	0	0
Service	sshLogin	1	3	0	2215	2219	0.125
	apacheStop	0	1	0	1711	1712	0.124
	mysqlWp	0	0	0	47	47	0.125
Kernel Modules	lsmod	0	0	0	251	251	0.045
	insmod	0	0	0	10	10	0.124
	rmmod	0	0	0	12	12	0.124
Docker	dockerHelloWorld	0	0	0	28	28	0.045
	dockerUbuntuLog	0	0	0	23	23	0.124
	dockerImages	0	0	0	1	1	0.001
	dockerPs	0	0	0	0	0	0
	dockerPSA	0	0	0	0	0	0
	dockerUbuntuSleep	0	0	0	2	2	0.001
	dockerRm	0	0	0	0	0	0
	dockerNginx	0	0	0	65	65	0.124
	dockerUbuntuBash	0	0	0	0	0	0
	dockerPrune	0	0	0	1	1	0.001
	dockerPruneVolumes	0	0	0	1	1	0.001
	dockerRmImages	0	0	0	2	2	0.001
	dockerUbuntuBashCp	0	0	0	0	0	0
	dockerUbuntuBashMv	0	0	0	18	18	0.124
	dockerUbuntuBashRm	0	0	0	3	3	0.08
	dockerUbuntuBashCat	0	0	0	24	24	0.044
Nextcloud	nextcloudStatus	0	1	0	2	3	0.001
	nextcloudAppList	0	1	0	43	44	0.124
	nextcloudUserList	0	1	0	2	3	0.001
	nextcloudUserAdd	0	1	0	102	103	0.045
	nextcloudGroupList	0	1	0	4	5	0.045

The blue line in Fig. 7 shows the accumulated frequency of the 20 most frequent system calls in our representative system call set. It can be seen that these 20 system calls make together nearly 90% of all system call calls. The

Table 4. Costs to trace system calls that are part of characteristic fingerprints ordered by the frequency in characteristic fingerprints. The descriptions are taken from the Linux Programmer's Manual [2].

System call	Description	Costs
`open`	Opens a file for reading or writing	0.0442
`stat`	Retrieves a files status	0.0791
`sendfile`	Transfers data between file descriptors	0.0003
`execve`	Executes a program	0.0009
`setfsuid`	Sets user identity used for filesystem checks	0.0000
`setfsgid`	Sets group identity used for filesystem checks	0.0000
`io_submit`	Submits asynchronous I/O blocks for processing	0.0000
`connect`	Initiates a connection on a socket	0.0007
`setreuid`	Sets a real and/or effective user or group ID	0.0000
`setregid`	Sets a real and/or effective user or group ID	0.0000
`bind`	Binds a name to a socket	0.0002
`creat`	Opens and possibly creates a file	0.0000
`chmod`	Changes permissions of a file	0.0000
`chroot`	Changes root directory	0.0000
`delete_module`	Unloads a kernel module	0.0000
`inotify_init1`	Initializes an inotify instance	0.0000
`finit_module`	Loads a kernel module	0.0000

red line shows the average characteristic fingerprint size when successively the system calls are removed from the characteristic fingerprints. One can spot two steps in the graph. One, when `stat` is not traced and one bigger when `open` is not traced. This is in accordance with Table 4 that shows only these two system calls are the "expensive". The average size of characteristic fingerprints shrinks by the factor 0.57.

Now we want to investigate in more detail how the sizes of characteristic fingerprints decreases when not tracing `stat` and `open`. Table 5 shows the size of characteristic fingerprints when tracing all system calls, all but not `stat`, all but not `open` and all but not `stat` and `open`. The table shows (as Fig. 7) that not tracing `stat` shrinks the size of characteristic fingerprints with an overage of about 20%. The same applies for `open` with about 27%. However, even though the table reveals quite large reduction rates are for some fingerprints, only two events have no vectors after the removal of `stat` and `open` from the characteristic fingerprints (Fig. 8).

The costs in Table 4 also show that not tracing `stat` and `open` should increase performance a lot. Table 6 gives an impression of the benefits in terms of performance of tracing fewer system calls. There, for three events the average overheads are calculated for different sets of traced system calls. Note, the "No Tracing"

Table 5. Loss of vectors in characteristic fingerprints when not tracing `stat` or `open` or both.

Class	Name	Before	w/o stat	w/o open	w/o stat,open
CLI	ls	1	1 (−0.00)	1 (−0.00)	1 (−0.00)
	cp	4	2 (−0.50)	3 (−0.25)	1 (−0.75)
	mv	2	1 (−0.50)	2 (−0.00)	1 (−0.50)
	cat	0	0 (−0.00)	0 (−0.00)	0 (−0.00)
	vmstat	6	6 (−0.00)	1 (−0.83)	1 (−0.83)
	netstat	15	15 (−0.00)	1 (−0.93)	1 (−0.93)
	tar	5	4 (−0.20)	5 (−0.00)	4 (−0.20)
	rm	1	1 (−0.00)	1 (−0.00)	1 (−0.00)
	shred	2	2 (−0.00)	1 (−0.50)	1 (−0.50)
	curl	1	1 (−0.00)	0 (−1.00)	0 (−1.00)
CLI Root	tailShadow	7	3 (−0.57)	7 (−0.00)	3 (−0.57)
	catCredentials	4	4 (−0.00)	3 (−0.25)	3 (−0.25)
	vimHosts	220	97 (−0.56)	127 (−0.42)	4 (−0.98)
	rmSudo	2	2 (−0.00)	2 (−0.00)	2 (−0.00)
	shredSudo	9	5 (−0.44)	8 (−0.11)	4 (−0.56)
Web	wordpressLogin	63	38 (−0.40)	41 (−0.35)	16 (−0.75)
	wordpressSearch	3	2 (−0.33)	2 (−0.33)	1 (−0.67)
	wordpressOpen	0	0 (−0.00)	0 (−0.00)	0 (−0.00)
Service	sshLogin	2219	1381 (−0.38)	1305 (−0.41)	467 (−0.79)
	apacheStop	1712	1697 (−0.01)	31 (−0.98)	16 (−0.99)
	mysqlWp	47	14 (−0.70)	35 (−0.26)	2 (−0.96)
Kernel Modules	lsmod	251	251 (−0.00)	1 (−1.00)	1 (−1.00)
	insmod	10	5 (−0.50)	8 (−0.20)	3 (−0.70)
	rmmod	12	6 (−0.50)	9 (−0.25)	3 (−0.75)
Docker	dockerHelloWorld	28	28 (−0.00)	3 (−0.89)	3 (−0.89)
	dockerUbuntuLog	23	12 (−0.48)	16 (−0.30)	5 (−0.78)
	dockerImages	1	1 (−0.00)	1 (−0.00)	1 (−0.00)
	dockerPs	0	0 (−0.00)	0 (−0.00)	0 (−0.00)
	dockerPSA	0	0 (−0.00)	0 (−0.00)	0 (−0.00)
	dockerUbuntuSleep	2	2 (−0.00)	2 (−0.00)	2 (−0.00)
	dockerRm	0	0 (−0.00)	0 (−0.00)	0 (−0.00)
	dockerNginx	65	45 (−0.31)	29 (−0.55)	9 (−0.86)
	dockerUbuntuBash	0	0 (−0.00)	0 (−0.00)	0 (−0.00)
	dockerPrune	1	1 (−0.00)	1 (−0.00)	1 (−0.00)
	dockerPruneVolumes	1	1 (−0.00)	1 (−0.00)	1 (−0.00)
	dockerRmImages	2	2 (−0.00)	2 (−0.00)	2 (−0.00)
	dockerUbuntuBashCp	0	0 (−0.00)	0 (−0.00)	0 (−0.00)
	dockerUbuntuBashMv	18	6 (−0.67)	13 (−0.28)	1 (−0.94)
	dockerUbuntuBashRm	3	1 (−0.67)	3 (−0.00)	1 (−0.67)
	dockerUbuntuBashCat	24	24 (−0.00)	0 (−1.00)	0 (−1.00)
Nextcloud	nextcloudStatus	3	3 (−0.00)	3 (−0.00)	3 (−0.00)
	nextcloudAppList	44	4 (−0.91)	43 (−0.02)	3 (−0.93)
	nextcloudUserList	3	3 (−0.00)	3 (−0.00)	3 (−0.00)
	nextcloudUserAdd	103	103 (−0.00)	17 (−0.83)	17 (−0.83)
	nextcloudGroupList	5	5 (−0.00)	3 (−0.40)	3 (−0.40)
Average		109	83 (−0.19)	38 (−0.27)	13 (−0.47)

may differ from the corresponding Δ values since Δ values were determined more pessimistically. While tracing all system calls can have an overhead of 90 which makes interactive usage nearly impossible, the overheads when only

Table 6. Average overhead of system call tracing.

Event name	No tracing	All syscalls	Only occurring	w/o `open`	w/o `stat`	w/o `open`, `stat`
tar	0.05 s	4.5 s (90)	0.2 s (4)	0.1 s (2)	0.18 s (3.6)	0.08 s (1.6)
sshLogin	0.5 s	21 s (42)	1.0 s (2)	0.6 s (1.2)	0.8 s (1.6)	0.6 s (1.2)
dockerHelloWorld	0.9 s	70 s (78)	8 s (8.9)	4 s (4.4)	6.5 s (7)	1.5 s (1.6)

tracing system calls that appear in characteristic fingerprints are a lot better. Removing `stat` and `open` from tracing has a huge impact. When not tracing `stat` and `open` the worst performance overhead we measure is 1.6.

7 Conclusion and Future Work

In this work we firstly were basically able to reproduce recent research [14] and could show that their kind of characteristic fingerprint calculation is also applicable for an even larger event set. We secondly showed that these characteristic fingerprints can be used to perform matching and so to actually reconstruct events based on log files (including system call traces). Thirdly we analyzed the possibilities and impact of system calls in characteristic fingerprints and show how to systematically reduce overhead by tracing only necessary system calls. We furthermore named discriminating system calls where future work can build upon.

Our measurements also revealed that with more events and so a bigger reference set there is more overlap between feature sets though. So, for some events we could not calculate a characteristic fingerprint. To calculate characteristic fingerprints it would be necessary to make the events more characteristic. This can be done by increasing the feature set or log source set. It would also be interesting how these characteristic fingerprints perform in term of unknown events. Future work should consider the false positive rate of unknown events.

Acknowledgements. We want to thank Felix Freiling for his valuable advice and feedback on this paper. This research was supported by the Federal Ministry of Education and Research, Germany, as part of the BMBF DINGfest project (https://dingfest.ur.de) and by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Research and Training Group 2475 "Cybercrime and Forensic Computing" (grant number 393541319/GRK2475/1-2019).

References

1. The xymon monitor. https://xymon.sourceforge.io/
2. Linux programmer's manual (2018). http://man7.org/linux/man-pages/dir_section_2.html
3. 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2019, Rotorua, New Zealand, 5–8 August 2019. IEEE (2019). https://ieeexplore.ieee.org/xpl/conhome/8883860/proceeding

4. Chabot, Y., Bertaux, A., Nicolle, C., Kechadi, T.: An ontology-based approach for the reconstruction and analysis of digital incidents timelines. Digit. Invest. **15**, 83–100 (2015)
5. Dewald, A.: Characteristic evidence, counter evidence and reconstruction problems in forensic computing. IT - Inf. Technol. **57**(6), 339–346 (2015). http://www.degruyter.com/view/j/itit.2015.57.issue-6/itit-2015-0017/itit-2015-0017.xml
6. Elasticsearch B.V.: Filebeat - lightweight shipper for logs (2020). https://www.elastic.co/products/beats/filebeat
7. Gerhards, R.: The syslog protocol. Technical report (2009)
8. Gladyshev, P., Enbacka, A.: Rigorous development of automated inconsistency checks for digital evidence using the B method. IJDE **6**(2) (2007). http://www.utica.edu/academic/institutes/ecii/publications/articles/1C35450B-E896-6876-9E80DA0F9FEEF98B.pdf
9. Gladyshev, P., Patel, A.: Finite state machine approach to digital eventreconstruction. Digit. Invest. **1**(2), 130–149 (2004). https://doi.org/10.1016/j.diin.2004.03.001
10. James, J.I., Gladyshev, P.: Automated inference of past action instances indigital investigations. Int. J. Inf. Sec. **14**(3), 249–261 (2015). https://doi.org/10.1007/s10207-014-0249-6
11. Khan, M.N.A., Chatwin, C.R., Young, R.C.D.: A framework for post-event timelinereconstruction using neural networks. Digit. Invest. **4**(3–4), 146–157 (2007). https://doi.org/10.1016/j.diin.2007.11.001
12. Kolosnjaji, B., Zarras, A., Webster, G., Eckert, C.: Deep learning for classification of malware system call sequences. In: Kang, B.H., Bai, Q. (eds.) AI 2016. LNCS (LNAI), vol. 9992, pp. 137–149. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50127-7_11
13. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 326–343. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39650-5_19
14. Latzo, T., Freiling, F.C.: Characterizing the limitations of forensic event reconstruction based on log files. In: 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering, TrustCom/BigDataSE 2019, Rotorua, New Zealand, 5–8 August 2019 [3], pp. 466–475 (2019). https://doi.org/10.1109/TrustCom/BigDataSE.2019.00069
15. Li, Z., et al.: UCLog: a unified, correlated logging architecture for intrusion detection. In: the 12th International Conference on Telecommunication Systems-Modeling and Analysis (ICTSM) (2004)
16. Marrington, A., Mohay, G.M., Morarji, H., Clark, A.J.: A model for computer profiling. In: ARES 2010, Fifth International Conference on Availability, Reliability and Security, 15–18 February 2010, Krakow, Poland, pp. 635–640 (2010). https://doi.org/10.1109/ARES.2010.95
17. Menges, F., et al.: Introducing DINGfest: an architecture for next generation SIEM systems. In: Langweg, H., Meier, M., Witt, B.C., Reinhardt, D. (eds.) Sicherheit 2018, Beiträge der 9. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), 25–27 April 2018, Konstanz. LNI, vol. P-281, pp. 257–260. Gesellschaft für Informatik e.V. (2018). https://doi.org/10.18420/sicherheit2018_21
18. Microsoft Corporation: Event logging (2019). https://docs.microsoft.com/en-us/windows/desktop/msi/event-logging

19. Ravi, S., Balakrishnan, N., Venkatesh, B.: Behavior-based malware analysis using profile hidden Markov models. In: 2013 International Conference on Security and Cryptography (SECRYPT), pp. 1–12. IEEE (2013)
20. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malwarebehavior using machine learning. J. Comput. Secur. **19**(4), 639–668 (2011). https://doi.org/10.3233/JCS-2010-0410
21. Taubmann, B., Kolosnjaji, B.: Architecture for resource-aware VMI-based cloud malware analysis. In: Proceedings of the 4th Workshop on Security in Highly Connected IT Systems, SHCIS@DAIS 2017, Neuchâtel, Switzerland, 21–22 June 2017, pp. 43–48 (2017). https://doi.org/10.1145/3099012.3099015
22. The Apache Software Foundation: Apache kafka - a distributed streaming platform (2020). https://kafka.apache.org/
23. Willems, C., Holz, T., Freiling, F.C.: Toward automated dynamic malware analysis using CWSandbox. IEEE Secur. Priv. **5**(2), 32–39 (2007). https://doi.org/10.1109/MSP.2007.45
24. Yadwadkar, N.J., Bhattacharyya, C., Gopinath, K., Niranjan, T., Susarla, S.: Discovery of application workloads from network file traces. In: FAST, pp. 183–196 (2010)
25. Yurcik, W., Abad, C., Hasan, R., Saleem, M., Sridharan, S.: UCLog+: a security data management system for correlating alerts, incidents, and raw data from remote logs. arXiv preprint cs/0607111 (2006)