FRAPpuccino: Fault-detection through Runtime Analysis of Provenance

Xueyuan Han, Thomas Pasquier, Tanvi Ranjan, Mark Goldstein, Margo Seltzer
Harvard University

Abstract

We present FRAPpuccino (or FRAP), a provenance-based fault detection mechanism for Platform as a Service (PaaS) users, who run many instances of an application on a large cluster of machines. FRAP models, records, and analyzes the behavior of an application and its impact on the system as a directed acyclic provenance graph. It assumes that most instances behave normally and uses their behavior to construct a model of *legitimate behavior*. Given a model of legitimate behavior, FRAP uses a *dynamic sliding window* algorithm to compare a new instance's execution to that of the model. Any instance that does not conform to the model is identified as an anomaly. We present the FRAP prototype and experimental results showing that it can accurately detect application anomalies.

1 Introduction

Platform as a service (PaaS) clouds have become increasingly popular for their efficient use of computational resources, providing users with an abstracted environment on which to easily deploy customized applications. Various market research companies estimate the growth of the PaaS market at about 30% annually for the next few years [28, 36]. However, PaaS cloud applications face two major challenges: 1) As an increasing number of businesses, enterprises, and organizations adopt cloud computing, cloud applications inevitably become a major target of cyber-attacks. For example, a recent DDoS attack against a top security blogger delivered by hijacked botnet was so aggressive that Akamai had to cancel the account [19]. According to the RightScale 2017 State of the Cloud Report [38], cloud security remains one of the top 5 challenges among cloud users; 2) PaaS clouds make it possible to build large-scale applications that can serve millions of users. A run-time fault introduced in an application can potentially render it useless. For example, a simple bug can repeatedly crash a server, making the service appear unavailable (§ 4).

We introduce **FRAP**puccino (**F**ault-detection through **R**untime **A**nalysis of **P**rovenance or FRAP for short), a fault/intrusion detection framework. FRAP detects anomalous behavior of cloud applications by using runtime provenance data to model correct program execution and detecting deviations from the model to identify potentially malicious behavior. Provenance describes system behavior as a labelled directed acyclic graph (DAG) representing interactions between system-level *entities* (e.g., file, sockets, pipes), *activities* (i.e., processes) and *agents* (i.e., users, groups). Provenance data can be abundant, so FRAP implements a streaming algorithm, using a *dynamic sliding window*, to avoid storing this data. From each running instance, the algorithm extracts a feature vector, which is a projection of the graph as a point into an *n*-dimensional space. We assume that most instances exhibit legitimate behavior so that clustering on these features will clearly divide the instances into good and bad sets. Thus, our application model includes two parts: the extracted features and the parameters of the clusters. Once FRAP has constructed such a model, it monitors program executions, extracts features from them, and reports any instances whose features deviate significantly from *good* behavior.

Unlike most behavioral-based intrusion/fault detection systems [29] that rely on system-call usage [14, 16, 41, 43, 47] to profile legitimate application behavior, FRAP uses provenance data that provides a more comprehensive view of program activities, including their effects on the underlying system. Prior research has shown that understanding the context of a program's execution, which the provenance DAG provides, leads to greater accuracy in detecting program anomalies [15, 40, 47]. Moreover, the provenance records provide data that can be analyzed to assist in root cause analysis, ideally providing actionable information. Our use of end-to-end provenance capture to detect intrusions or faults differs from other end-to-end tracing approaches in two major ways: 1) by using runtime graphical and statistical analysis on provenance DAGs of normal instances of an application, FRAP requires no application instrumentation or annotation, while systems such as Pip [37] need developer-provided specifications of expected behavior; 2) FRAP analyzes interactions between potentially all executing applications and the system as naturally presented by provenance DAGs, while systems such as Magpie [6] and SpectroScope [39] use event logs, which represent a carefully curated subset of system activity, which may or may not capture the key actions. Our goal is to show an alternative approach and evidence of its efficacy in tackling a long-standing problem of intrusion/fault detection.

The contributions of this work are: 1) a novel approach that combines provenance and graphical and statistical analysis to model the behavior of cloud applications; 2) a *dynamic sliding window* algorithm that allows efficient processing of large provenance data to achieve online detection; and 3) an implementation of our framework with demonstrated accuracy.

2 Background

We build FRAP using two existing open-source tools: 1) CamFlow [33, 34], a state-of-the-art provenance capture system; and 2) GraphChi [2, 26], a highly-efficient graph processing framework. However, the concepts are not tied to either implementation.

CamFlow [34]: Provenance records the chronology of ownership, change, and movement of an object or a resource. We use provenance data to understand the interactions between the monitored application, other applications, and the underlying operating system. There are many provenance capture systems available, including PASS [31], Hi-Fi [35], Linux Provenance Module [7], and CamFlow [34]. We chose to use CamFlow, because it tracks multiple applications, their interactions with the system, and their interactions with each other. Moreover, it both limits the amount of information captured (i.e., you can specify which applications to trace) and ensures completeness by propagating capture to any programs that a traced application invokes. The capture is built upon Linux Security Module (LSM) hooks that provide completeness guarantees [13, 17, 22]. CamFlow also provides a facility to conveniently stream the provenance data captured through messaging middleware such as MQTT[3], RabbitMQ[4], or Apache Flume[1].

GraphChi [26]: Provenance data is naturally represented as a DAG in which each node represents an entity, an activity, or an agent, and each directed edge represents an interaction between two nodes. For example, a file *was generated by* a process, or a process *used* a packet. Our framework uses type information in a provenance record as labels to construct a labeled DAG. DAGs can be efficiently processed by graph processing engines. We chose GraphChi, a vertex-centric graph processing model, to generate program models and to detect anomalies. GraphChi uses a *parallel sliding window* method to achieve efficient computation of vertices.

3 The FRAPpuccino Framework

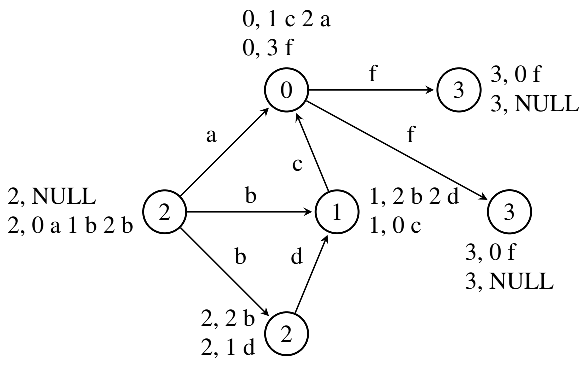
The FRAP framework consists of three stages: 1) the *learning stage* determines the size of the *dynamic sliding window* and creates a model of correct program behavior; 2) the *detection stage* periodically compares instances of a program's execution with the model, notifying the user upon detection of unusual program behavior; and 3) the *revision stage* improves the model by incorporating additional information when we encounter a false positive.

FRAP starts with the learning stage, iterates through the detection stage until it needs to revise the model, and transitions back to the detection stage after the revision.

3.1 Learning Stage

The learning stage analyzes the provenance DAG of each instance of a program, creating a model to describe its legitimate behavior. Provenance data can grow infinitely large, making it impossible to analyze them as a whole. However, past research [16, 21] has shown that a program usually has a limited set of interactions with the system (e.g., writing to a file, sending a packet), repeating them in different orders as it executes. Therefore, we claim that one can learn most of a program's behavior from a subset of its provenance data. We begin with an overview of our approach, followed by a more detailed discussion of each step. FRAP uses a simple but effective algorithm to determine the number of consecutive provenance records it needs to examine to create a program model, which is the *dynamic sliding window* size (used in the detection stage). Using this subset of records, FRAP transforms the provenance DAG into a multidimensional numerical feature vector. We construct this feature vector in three steps: First, we run a label propagation algorithm that constructs a label for each node, representing the structure of the graph around the node. Second, we count the number of instances of each unique label. Third, we construct a feature vector consisting of all the label counts. A feature vector is therefore the result of dimensionality reduction, abstracting a program's behavior into numerical values. Finally, FRAP clusters the feature vectors with the goal of grouping all well-behaved instances together and leaving badly behaved instances in different clusters. Thus, our model consists of the feature vectors of well behaved instances and the parameters (centroid and radius) of the clusters in which they reside.

Determining the Dynamic Sliding Window Size: The goal of finding a dynamically sized window is to minimize the number of provenance records needed to characterize an application. We determine the size during capture by maintaining two counts. The first counts every edge examined until we declare a window size. We

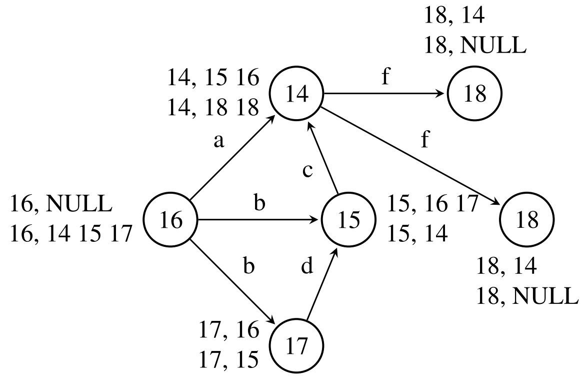


(a) Before relabeling

Relabeling Map

0, 1 c 2 a → 4	0, 3 f → 5	4, 5 → 14
1, 2 b 2 d → 7	1, 0 c → 6	7, 6 → 15
2, NULL → 8	2, 0 a 1 b 2 b → 9	8, 9 → 16
2, 2 b → 11	2, 1 d → 10	11, 10 → 17
3, 0 f → 13	3, NULL → 12	13, 12 → 18

(b) Populating Relabeling map



(c) After relabeling

Figure 1: (a) Before the first iteration, we label each node with its own label, its neighboring nodes' labels, and its incident edges' labels. (b) Each node inserts its sorted in-edge neighbor label list and its sorted out-edge neighbor label list into the relabeling map, and gets a new label based on these two label lists. In-edge node relabeling is shown in red, out-edge node relabeling is shown in blue, and final relabeling is shown in black. (c) After relabeling, nodes have both new identities and new labels while edge labels are unchanged.

calculate the second by the following process: We examine each edge and assign it a triple consisting of the original edge type and the types of each vertex. We count the number of edges processed until we encounter a triple we have never seen before, at which point we reset the count to 0. If the first counter reaches an implementation-defined threshold (§ 4), or if the second counter reaches a user-defined threshold, we set the dynamic window size to be the value of the first counter.

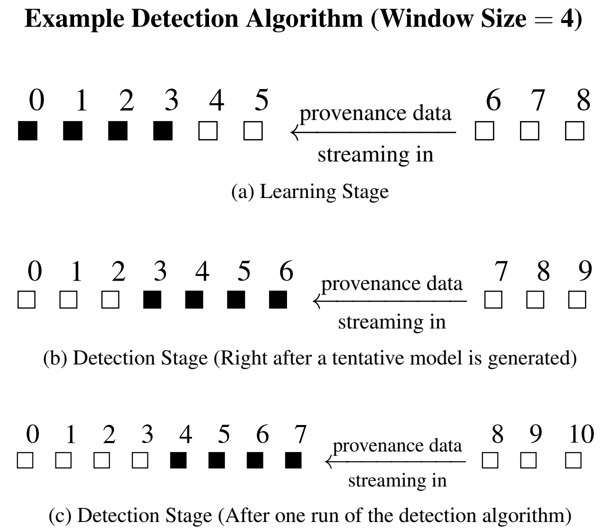
Generating a Program Model: To generate a model, FRAP computes a feature vector for each instance, clusters those vectors, and discards the vectors in isolated clusters. FRAP generates a feature vector, using the following iterative algorithm: relabel each vertex in the DAG using the current vertex label and the labels of its incoming and outgoing neighbors. During the first iteration, we also incorporate the labels of a vertex's incident edges (Fig. 1), but need not include these in later iterations, because that information is already encoded in the labels of the neighboring vertices. After each iteration, new labels encode longer sequences of interactions between the program and the system. FRAP generates a vector containing counts of all seen labels, including the ones from previous iterations, and then clusters the vectors from all instances. We empirically determine that four iterations produces the best results. To cluster, FRAP uses symmetric Kullback-Leibler divergence [25] or *Kullback-Leibler Distance* (KLD) with *back-off probability* [30] as the distance metric to measure the similarity between two feature vectors. KLD has been used, for example, in statistical language modeling [12] and text categorization [8]. We use two applications of K-means clustering: first we cluster on distances between feature vectors, which helps us select K for the second K-means clustering, which computes the actual model.

We assume that clusters containing many vectors represent legitimate behavior and want to discard feature vectors in clusters isolated from these good clusters. Wagstaff et al. [45] have shown that cluster accuracy improves when additional information is available to the problem domain. We use our assumption that most instances are well-behaved as this additional information. Specifically, we hypothesize that there exists an observable difference between *inter-cluster* and *intra-cluster* distances. First we set K equal to the total number of instances we are analyzing and run K-means clustering on the *pairwise KL distance between each pair of instances*. We then set K to the number of populated clusters and run a second K-means clustering on the feature vectors themselves. This produces our model consisting of the set of feature vectors in clusters containing more than one instance and the parameters of those clusters (e.g., centroid and radius).

3.2 Detection Stage

FRAP monitors an instance by taking its provenance data from a window of execution, generating a feature vector as in § 3.1, and checking whether this vector fits into any clusters by comparing the distance between the feature vector and the centroids of the clusters. An instance is considered abnormal if it does not fit into any of the model's clusters.

Dynamic Sliding Window: FRAP uses a *dynamic sliding window* approach to continuously monitor an instance while it runs uninterrupted. FRAP only stores and analyzes the provenance DAG within this window. Once



(a) Learning Stage

(b) Detection Stage (Right after a tentative model is generated)

(c) Detection Stage (After one run of the detection algorithm)

Figure 2: Black rectangles represent the provenance records in the window. White ones to the left of the arrow are captured but not yet processed, and to the right will be generated in the future.

it determines that the DAG is part of a normal program execution, FRAP can safely discard the data. The size of the window is determined in the learning stage (§ 3.1). As shown in Fig. 2, records $0,1,2,3$ in 2a are used to generate a model while new provenance data (records 4 and 5) are streaming in. The window slides each time FRAP finishes running the detection algorithm (described in the next paragraph), to include new records for the next round of analysis (2b and 2c). Since the window slides incrementally, each record appears in *many* different windows, maintaining a holistic view of program execution. New records may contain vertices related to ones processed in previous windows. This overlapping preserves a vertex's n-hop neighbors in which n becomes larger as the overlap increases. This means that a larger number of iterations becomes more meaningful, since longer sequences of program-system interactions can be preserved.

Detection Algorithm: Given a feature vector of a window of execution, FRAP uses the same distance metrics (i.e., KLD) to see if the instance lies in an existing cluster (i.e., the distance between it and the centroid is smaller than the cluster's radius). If it does not lie in an existing cluster, FRAP re-runs (the second) K-means clustering, and only then reports the instance as an anomaly if it still does not lie within a cluster of legitimate executions. Since the detection algorithm is entirely local between an instance and the model, the computation is parallelizable and scalable.

3.3 Revision Stage

FRAP identifies and collects false-positive instances and re-clusters, including them in the model. This reclustering is fast, which is important, because instances are running concurrently. A fast transition from the revision stage to the detection stage ensures maximal overlap between the windows before and after revision (§ 3.2).

Our current false-positive identification is somewhat crude. We assume that most instances are correct and revise the model only when detecting a large number of abnormal instances. However, this assumption does not always hold. For example, a corrupted database may cause many client requests to fail. FRAP should not learn such behavior and include it in the model. Another problematic scenario is when a program has normal behavior (A, B, C), and the model contains only one behavior (A). When many instances are displaying behavior B and one instance is displaying C, then only B will be included in the model while that one instance will be considered unusual. Developing a more principled approach to this problem is left for future work.

4 Implementation and Results

We describe our implementation [20], experimental results and future work.

Implementation-specific Window Size: When we constantly receive unique provenance records, our implementation-defined threshold determines the window size (§ 3.1). GraphChi uses a parallel sliding window method to perform computation on the vertices inside the window in memory, write the results to disk, and move the window to load the next set of vertices to memory. To avoid high-latency I/O operations, the learning stage takes the provenance data of the maximum size of the memory for each instance and allows GraphChi to perform computation on all vertices in-memory.

Program Models and Feature Vectors: GraphChi performs vertex relabeling to generate a feature vector for each instance. Our implementation uses a global map to achieve consistent relabeling. We need to handle GraphChi's asynchronous model of computation, in which an update of a vertex label is immediately visible to its neighbors in the same iteration. This means if a vertex runs its computation *after* one of its neighbors updates itself, it will take that neighbor's updated label, which should be used in the next iteration. We solve this problem by alternating the computation with an *update phase* and a *swap phase*. In the update phase (which consists of one iteration), each vertex computes its new label without broadcasting it to its neighbors. Therefore, within the same iteration, all vertices take their neighboring vertices' labels from the previous update phase. The next iteration is the swap phase, where all vertices broadcast their new labels to their neighbors so that in the next update phase, they can all read the latest labels.

Preliminary Results: We conducted a number of experiments to see if FRAP is able to capture instances with unusual behavior in a pool of well-behaved instances. We used both Hellinger distance [32] and Euclidian dis-

Metrics	Captured Bad Instance (Learning)	Captured Bad Instance (Detection)
KLD	Yes	Yes
Hellinger	No	No
Euclidean	Yes	Yes

Table 1: This experiment runs 10 clients sending requests to the server, one of which causes the server to behave abnormally during learning. The same bad behavior occurs again during detection.

tance, in addition to KLD, to see how different similarity metrics affect FRAP's performance. Table 1 shows the results of one of these experiments. In this experiment, we set up a Ruby server in a simulated cloud environment. The server handles requests from multiple clients, and causes an out-of-memory server crash[18] – a known system level Ruby vulnerability – for certain URLs.

We see that all but Hellinger distance are able to identify the badly behaved Ruby instance. Hellinger distance does not work well, because it always produces values between 0 and 1, which make it difficult for K-means clustering to create meaningful clusters. Hellinger consistently placed all instances in a single cluster. After manual inspection, we also discovered that while Euclidean distance successfully identified the badly behaved instance, it mistakenly considered two normal instances as abnormal during the learning stage (i.e., false negatives), resulting in a slightly less accurate model than KLD. Appendix A provides a reference to the current prototype and other experimental datasets.

Future Work: We plan to develop more sophisticated ways to identify false-positives as outlined in § 3. We also want to further optimize our current implementation and identify situations in which more sophisticated learning algorithms will improve accuracy. One important area of optimization is our global relabeling map. Since GraphChi processes vertices in parallel, we need to maintain map consistency and avoid race conditions. Our current implementation is a single-point of contention and can easily be a performance bottleneck. We also have to garbage collect the map to keep its size manageable. We also want to experiment other algorithms to improve our clustering. For example, Principle Component Analysis [23] might help us further reduce dimensionality of feature vectors and discard misleading features.

Our current prototype does not have all the pieces integrated, so we used manual intervention to achieve the end to end pipeline. In particular, we do not directly stream the provenance to the analyzer. Instead, we capture the provenance and then run the analyzer over the provenance stream. Additionally, we have not yet integrated the revision stage. These will both be available in the next release of the software.

5 Related Work

A number of systems use sequences of system calls to detect program/system anomalies. pH [43] uses temporally proximate system call sequences to model the behavior of a program. More recent systems have proposed more advanced analyses. For example, MaMaDroid [27] uses static program analysis to obtain a program's call graph and dynamically builds a Markov-chain [24] model of the graph during runtime. The feature vector of their program model consists of the probabilities of each state transition in the Markov chain. CMarkov [47] includes calling context of system calls when performing static program analysis, which further refines their Markov model. FRAP uses provenance to build a program model without the help of static program analysis. Its relabelling mechanism concisely encodes system call sequences and their contexts. Moreover, FRAP can detect anomalies in complex applications composed of multiple processes including distributed ones by capturing, aggregating, and analyzing provenance data from multiple machines. It is not restrained by per-process system call sequences. We leave comparing FRAP performance with that of other detection systems for future work.

Similar to FRAP, systems such as Magpie [6], Pinpoint [11], Pip [37] and SpectroScope [39] use end-to-end tracing to detect anomalies that could indicate bugs. However, unlike FRAP, they all capture request flows within distributed systems and analyze event logs either on a per-event basis or on whole paths. More importantly, although these systems are able to infer bugs, they are mainly designed to diagnose performance problems, not intrusions, in distributed systems. Moreover, Pinpoint and Pip require manually annotating applications, an error-prone and significant burden on developers, while FRAP can analyze all applications as long as the underlying system captures provenance.

For systems that do not have detailed end-to-end tracing capabilities, some black-box diagnosis techniques have been proposed, e.g., using message send/receive events to deploy black-box performance debugging [5].

Our graph analysis is related to work on graph kernels [44]. They are widely used in studying relationships between structured graphs [44]. In particular, our relabeling algorithm is based on the subtree Weisfeiler-Lehman graph kernel [42] and is a variation of the Weisfeiler-Lehman test of isomorphism [46].

6 Conclusion

We present a novel approach to detecting unusual behavior in programs running on PaaS clouds and demonstrate its usability via our implementation. We believe current advances in provenance capture systems open a new landscape for research in cloud computing and computer systems.

Discussion Topics

We assume that the provenance data we capture are trustworthy and that attackers cannot modify provenance data to mask the application's execution trail. What should we do if this assumption does not hold in practice? Our system requires mitigation techniques to guard against non-trustworthy provenance data or detect provenance data tampering as a different form of intrusion. This problem has been explored in the literature [7, 9], and Zhou et al. [48] presented a solution to secure network provenance. Can we simply "plug-and-play" those mechanisms in our system? How do we comprehensively secure various sources of provenance used in our system?

We propose that user involvement can help build a better model by allowing users to identify false-positives. What should we do to provide users with *meaningful* provenance information to assist their judgement? One possible solution is to apply differential provenance [10] to explain the sources of anomalies. However, differential provenance has only been applied to network provenance. How do we apply this technique to other domains?

There are a variety of intrusion detection systems (IDS) for the cloud environment. Modi et al. [29] categorized them into eight different techniques, identifying both their strengths and weaknesses. FRAP is a behavioral-based detection system, but unlike other systems in this category, it uses provenance to model the behavior of an application. From what aspects does FRAP work better than other behavioral-based detection systems and than other cloud IDS's at large? What kinds of intrusions are intrinsically hard for FRAP to detect but easy for other IDS's?

Acknowledgements

We thank Daniel Margo for inspiring discussions of graph processing frameworks. This work was supported by the US National Science Foundation under grant SSI-1450277 End-to-End Provenance.

References

[1] Apache flume. `https://flume.apache.org`.

[2] Graphchi. `https://github.com/GraphChi`.

[3] Mqtt. `http://mqtt.org`.

[4] Rabbitmq. `https://www.rabbitmq.com`.

[5] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 74–89.

[6] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using magpie for request extraction and workload modelling. In *OSDI* (2004), vol. 4, pp. 18–18.

[7] BATES, A. M., TIAN, D., BUTLER, K. R., AND MOYER, T. Trustworthy whole-system provenance for the linux kernel. In *Usenix Security* (2015), pp. 319–334.

[8] BIGI, B. Using kullback-leibler distance for text categorization. In *European Conference on Information Retrieval* (2003), Springer, pp. 305–319.

[9] BRAUN, U., SHINNAR, A., AND SELTZER, M. I. Securing provenance. In *HotSec* (2008).

[10] CHEN, A., WU, Y., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. Differential provenance: Better network diagnostics with reference events. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks* (2015), ACM, p. 25.

[11] CHEN, Y.-Y. M., ACCARDI, A., KICIMAN, E., PATTERSON, D. A., FOX, A., AND BREWER, E. A. Path-based failure and evolution management.

[12] DAGAN, I., LEE, L., AND PEREIRA, F. C. Similarity-based models of word cooccurrence probabilities. *Machine learning 34*, 1-3 (1999), 43–69.

[13] EDWARDS, A., JAEGER, T., AND ZHANG, X. Runtime verification of authorization hook placement for the linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (2002), ACM, pp. 225–234.

[14] FENG, H. H., GIFFIN, J. T., HUANG, Y., JHA, S., LEE, W., AND MILLER, B. P. Formalizing sensitivity in static analysis for intrusion detection. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on* (2004), IEEE, pp. 194–208.

[15] FENG, H. H., KOLESNIKOV, O. M., FOGLA, P., LEE, W., AND GONG, W. Anomaly detection using call stack information. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on* (2003), IEEE, pp. 62–75.

[16] FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on* (1996), IEEE, pp. 120–128.

[17] GANAPATHY, V., JAEGER, T., AND JHA, S. Automatic placement of authorization hooks in the linux security modules framework. In *Proceedings of the 12th ACM conference on Computer and communications security* (2005), ACM, pp. 330–339.

[18] GAZIEV, A. How ruby 2.2 can cause an out-of-memory server crash, 2015. `https://evilmartians.com/chronicles/ruby-2_2-oom`.

[19] GREENE, T. Largest ddos attack ever delivered by botnet of hijacked iot devices, 2016. `http://www.networkworld.com/article/3123672/security/largest-ddos-attack-ever-delivered-by-botnet-of-hijacked-iot-devices.html`.

[20] HAN, X. michael-hahn/frap: v1.1.1, 2017. DOI:10.5281/zenodo.571444, `https://github.com/michael-hahn/frap`.

[21] HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. Intrusion detection using sequences of system calls. *Journal of computer security 6*, 3 (1998), 151–180.

[22] JAEGER, T., EDWARDS, A., AND ZHANG, X. Consistency analysis of authorization hook placement in the linux security modules framework. *ACM Transactions on Information and System Security (TISSEC) 7*, 2 (2004), 175–205.

[23] JOLLIFFE, I. *Principal component analysis*. Wiley Online Library, 2002.

[24] KEMENY, J. G., SNELL, J. L., ET AL. *Finite markov chains*, vol. 356. van Nostrand Princeton, NJ, 1960.

[25] KULLBACK, S., AND LEIBLER, R. A. On information and sufficiency. *The annals of mathematical statistics 22*, 1 (1951), 79–86.

[26] KYROLA, A., BLELLOCH, G. E., GUESTRIN, C., ET AL. Graphchi: Large-scale graph computation on just a pc. In *OSDI* (2012), vol. 12, pp. 31–46.

[27] MARICONTI, E., ONWUZURIKE, L., ANDRIOTIS, P., DE CRISTOFARO, E., ROSS, G., AND STRINGHINI, G. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433* (2016).

[28] MARKO, K. Cloud-first application platforms ? paas tools to watch in 2017, 2016. `http://diginomica.com/2016/12/30/cloud-first-application-platforms-paas-tools-watch-2017/`.

[29] MODI, C., PATEL, D., BORISANIYA, B., PATEL, H., PATEL, A., AND RAJARAJAN, M. A survey of intrusion detection techniques in cloud. *Journal of Network and Computer Applications 36*, 1 (2013), 42–57.

[30] MORI, R. D. *Spoken dialogues with computers*. Academic Press, Inc., 1997.

[31] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track* (2006), pp. 43–56.

[32] NIKULIN, M. S. Hellinger distance. *Encyclopedia of Mathematics* (2001).

[33] PASQUIER, T. Camflow/camflow-dev, 2017. DOI:10.5281/zenodo.571427, `https://github.com/CamFlow/camflow-dev`.

[34] PASQUIER, T. F.-M., SINGH, J., BACON, J., AND EYERS, D. Information flow audit for paas clouds. In *Cloud Engineering (IC2E), 2016 IEEE International Conference on* (2016), IEEE, pp. 42–51.

[35] POHLY, D. J., MCLAUGHLIN, S., MCDANIEL, P., AND BUTLER, K. Hi-fi: collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), ACM, pp. 259–268.

[36] RESEARCH, Z. M. Global public cloud platform as a service (paas) market worth usd 9.12 billion by 2021 - zion market research, 2016. `https://globenewswire.com/news-release/2016/09/26/874593/0/en/Global-Public-Cloud-Platform-as-a-Service-PaaS-Market-worth-USD-9-12-billion-by-2021-Zion-Market-Research.html`.

[37] REYNOLDS, P., KILLIAN, C. E., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006), vol. 6, pp. 115–128.

[38] RIGHTSCALE. Largest ddos attack ever delivered by botnet of hijacked iot devices, 2017. `http://assets.rightscale.com/uploads/pdfs/RightScale-2017-State-of-the-Cloud-Report.pdf`.

[39] SAMBASIVAN, R. R., ZHENG, A. X., DE ROSA, M., KREVAT, E., WHITMAN, S., STROUCKEN, M., WANG, W., XU, L., AND GANGER, G. R. Diagnosing performance changes by comparing request flows. In *NSDI* (2011), pp. 43–56.

[40] SEKAR, R., BENDRE, M., DHURJATI, D., AND BOLLINENI, P. A fast automaton-based method for detecting anomalous program behaviors. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on* (2001), IEEE, pp. 144–155.

[41] SHARIF, M., SINGH, K., GIFFIN, J., AND LEE, W. Understanding precision in host based intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection* (2007), Springer, pp. 21–41.

[42] SHERVASHIDZE, N., SCHWEITZER, P., LEEUWEN, E. J. V., MEHLHORN, K., AND BORGWARDT, K. M. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research 12*, Sep (2011), 2539–2561.

[43] SOMAYAJI, A. B. *Operating system stability and security through process homeostasis*. PhD thesis, The University of New Mexico, 2002.

[44] VISHWANATHAN, S. V. N., SCHRAUDOLPH, N. N., KONDOR, R., AND BORGWARDT, K. M. Graph kernels. *Journal of Machine Learning Research 11*, Apr (2010), 1201–1242.

[45] WAGSTAFF, K., CARDIE, C., ROGERS, S., SCHRÖDL, S., ET AL. Constrained k-means clustering with background knowledge. In *ICML* (2001), vol. 1, pp. 577–584.

[46] WEISFEILER, B. J., AND LEMAN, A. A reduction of a graph to a canonical form and an algebra arising during this reduction, nauchno–technicheskaja informatsia, 9 (1968), 12–16.

[47] XU, K., TIAN, K., YAO, D., AND RYDER, B. G. A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on* (2016), IEEE, pp. 467–478.

[48] ZHOU, W., FEI, Q., NARAYAN, A., HAEBERLEN, A., LOO, B. T., AND SHERR, M. Secure network provenance. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 295–310.

A Availability

The work presented in this paper is open-source and available for download at

`https://github.com/michael-hahn/frap`