Intrusion Detection using Parzen-Windows on Provenance Graph Statistics

Kenny Yu
Harvard University
kennyyu@college.harvard.edu

R. J. Aquino
Harvard University
rjaquino@college.harvard.edu

CS261, Fall 2013

Abstract

Provenance is data that describes how a digital artifact came to be in its current state. We hypothesize that intrusions on a system leave behind anomalies in the lineage of digital artifacts. We present an intrusion detection approach to find these anomalies by analyzing centrality metrics on provenance graphs. We use a Parzen-Window approach (TODO CITE) on various provenance graph centrality metrics (TODO CITE) to determine probability density estimates of normal behavior, and we use these density estimates to determine if an intrusion occurred. We used this approach to analyze *user-to-remote* (u2r) intrusions and *remote-to-local* (r2l) intrusions (TODO: include r2l?) from the 1998 DARPA Intrusion Detection data sets (TODO CITE) and achieved up to *TODO true positive rate for intrusions* accuracy in detecting intrusions with only *TODO false positive rate for intrusions* accuracy. We also present future work to extend our intrusion model to an online intrusion detection system.

1 Introduction

For as long as systems have existed, there have been malicious users that attempt to exploit vulnerabilities in systems to gain unintended privileged access. As a result, system designers and administrators place a large effort in securing systems and preventing intrusions. However, intrusions inevitably occur because of bugs or flaws within the system, and as a result, system administrators want to have some automatic way of detecting these intrusions.

Intrusions often leave behind digital artifacts, e.g., unusual output or log files, a process executed with an unusual arguments or environment, unusual system call activity by a process. In addition to unusual digital artifacts, we hypothesize that intrusions also leave behind abnormalities in the *provenance* of these digital artifacts, the lineage of how these digital artifacts were created.

Provenance is data that describes how digital artifacts came to be in their current state. Provenance data is typically structured as a directed acyclic graph with typed nodes and typed edges. Nodes typically include processes and files, and edges are directed from nodes to their dependencies. For example, process nodes have edges directed towards input file nodes and to the parent process's node, and file nodes have edges directed towards previous versions of the file and to the nodes of the processes that modified the file.

Existing intrusion detection systems (IDS) use provenance data only to a limited extent. ??? person analyzes basic statistics on provenance graphs (e.g., number of nodes, number of edges, ... TODO) in order to facilitate easier manual intrusion detection by a human (CITE WORK). Their work, however, does not present a way of using provenance graph to automatically determine intrusions. Other systems use provenance data to determine the scope of an intrusion. For example, Backtracker (CITE WORK) uses provenance graphs to build causality graphs of intrusions: once the system has determined an intrusion has occurred, the system follows the causality graph backwards determine the original source of the attack, and then it follows the causality graph forwards to determine all the objects tainted by the attack. However, the authors do not present a way of using provenance data to automatically detect intrusions.

¿¿¿ INSERT PROVENANCE GRAPH SMALL EXAMPLE HERE SHOWING DATA FLOW

In this paper, we present an approach to use provenance data to automatically detect intrusions. Intrusion detection can be framed as a *novelty detection* problem, in which one attempts to decide whether an unknown test pattern is produced by an underlying distribution corresponding to a training set of nor-

mal patterns (CITE WORK). However, ???authors note that in the case of intrusion detection, novel or abnormal patterns are typically difficult to obtain, and as a result, they present a *Parzen-Window* approach for non-parametric density estimation. Using this technique, they obtain a high degree of success in identifying various intrusion types (CITE WORK). Because obtaining abnormal patterns is difficult, we borrow their Parzen-Window approach to build models of normal behavior using various provenance graph statistics and graph centrality metrics, and we evaluate the success of this technique on *user-to-local* (u2l) intrusions (intrusions that provide unauthorized access to local superuser (root) privileges), and *remote-to-local* (r2l) intrusions (intrusions that provide unauthorized access from a remote machine).

The main contributions of this paper are the following:

1. Discuss which provenance graph statistics we chose to use, how we chose them, and why we chose them.

2. Analyze the Parzen-Window technique with various provenance graph statistics on a real intrusion detection data set and evaluate its accuracy.

3. Discuss the limitations of the approach and present future work to transform the technique into an online intrusion detection system.

POSSIBLY INCLUDE THIS * the papers suck at finding u2r * because they don't look at provenance graph structure * we can do better?

2 Related Work

2.1 Existing IDSs using Provenance

Many existing intrusion detection systems use provenance in some limited capacity; however, none of them have used provenance for a fully automated detection system. Somayaji and Forrest's work in analyzing sequences of system calls for intrusion detection (CITE WORK) can be seen as one of the earliest works in building an intrusion detection system using provenance: sequences of system calls can be seen as a limited form of provenance, as many systems build provenance graphs from system call traces (CITE SPADE AND BURRITO). By analyzing the sequences of system calls a process makes over time, their system builds a profile of "normal" process behavior. When the process in the future makes enough patterns of unrecognized sequences of system calls, the system flags this process as behaving abnormally and attempts to stop the process, either through exponentially slowing down system calls or aborting system calls entirely. Notably, their system is trained only by analyzing "normal" data without knowing explicitly what is considered "abnormal." Because their system achieves high accuracy in determining intrusions with only sequences of system calls, we believe that having real provenance data in a graph structure–which in theory would be a superset of system call data–would allow us to achieve comparable, if not better intrusion detection accuracy.

More recent work make use of the structure of provenance as a directed graph for a semi-automatic form of intrusion detection, and to determine the scope of an intrusion. ??? developed Backtracker (CITE) a modified Linux kernel that tracks dependencies between operating system objects (files, processes, file names) similar to provenance graphs. Backtracker builds a backwards causal graph to determine the entry point of an intrusion, and it builds a forward causal graph to determine possibly tainted files, processes, or hosts in a distributed system. However, Backtracker does not provide a way of automatically determining if an intrusion occurred using provenance data; the provenance graph structure is only utilized after-the-fact.

??? make use of simple provenance graph statistics to categorize provenance graphs and to assist a human in manual intrusion-detection. Given a data set of many provenance graphs, they calculated basic statistics on the graphs (e.g., total nodes, total edges, max incoming edges on a node, max outgoing edges on a node, node/edge ratio, average total edges per node, etc.). Their approach allows a human to more easily notice anomalies in provenance graphs, but they do not provide a way to automate this detection. We borrow their ideas of using simple provenance graph statistics as features to detect intrusions.

These works demonstrate that provenance data is indeed useful in collecting information about intrusions, and Somayaji and ???'s work suggests that it might be possible to build a fully automated intrusion detection system by making use of the graph structure of provenance.

2.2 Parzen-Windows

To address the difficulty of obtaining abnormal patterns for novelty detection on intrusion detection, we borrow the Parzen-Window model approach proposed by ???. ??? person attempt to solve the lack of abnormal patterns problem by using Parzen-

Windows to build *nonparametric* density estimations of normal behavior (CITE). A density estimation is considered *nonparametric* if the estimation makes no assumptions about the forms of the PDFs, except that PDFs are smooth. By using a nonparametric density estimation, one can build a probability density estimate of normal behavior by only having examples of normal behavior and no examples of abnormal behavior.

A *Parzen-window* estimate of a probability density function $p(\mathbf{x})$ based on n examples in a dataset D drawn from model $\mathcal{M}$ is given by:

$$p(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n}\delta_n(\mathbf{x}-\mathbf{x_i})$$

where $\delta_n(\cdot)$ is a kernel function. They chose to use radially-symmetric Gaussian kernel functions because Gaussian functions are smooth and therefore $p(\mathbf{x})$ will be smooth, and because a radially-symmetric Gaussian function can be specified by a single parameter, the variance of the kernel. Using a common variance σ^2 for all the Gaussian kernels, we can rewrite $p(\mathbf{x})$ as:

$$p(\mathbf{x}) = \frac{1}{n(2\pi)^{d/2}\sigma^d}\sum_{i=1}^{n}\exp\left\{-\frac{||\mathbf{x}-\mathbf{x_i}||^2}{2\sigma^2}\right\}$$

where d is the dimensionality of the feature space $\mathbf{x}$.

Let ω_1 denote the state of being normal and ω_0 the state of being abnormal. To test if an example $\mathbf{x} \in \omega_1$, they reframe the problem using hypothesis testing. Let $\mathbf{y}$ be an arbitrary example from D drawn from $\mathcal{M}$, and let

$$L(\mathbf{y}) = \log p(\mathbf{y})$$

be the log-likelihood of $\mathbf{y}$ with respect to $\mathcal{M}$. Then we test the hypothesis that $L(\mathbf{x})$ is drawn from the distribution of the log-likelihood of the random examples in D with:

$$\begin{aligned} P(L(\mathbf{y}) \leq L(\mathbf{x})) &= \frac{\#\mathbf{y} \text{ with } L(\mathbf{y}) \leq L(\mathbf{x})}{n} \\ &> \psi \end{aligned}$$

for some threshold $0 < \psi < 1$, called the *false detection rate*. Thus, $\mathbf{x} \in \omega_1$ is behaving normally if and only if $P(L(\mathbf{y}) \leq L(\mathbf{x})) > \psi$.

In this paper, we use this same model to build profiles of "normal" behavior for each process. We use one-dimensional feature vectors x, and we use various provenance graph statistics and graph centrality metrics as our features. We describe the various statistics and graph centrality metrics we explored in the following section. For the common variance of our Gaussian kernels, we chose the variance to be the variance of the data set D for each unique process name. We vary the features we used and values of ψ and evaluate its performance on an intrusion detection data set.

2.3 Provenance Graph Features

We were inspired by the work proposed by ??? who use simple provenance graph statistics to aide in manual intrusion detection by humans. We extend their idea to make use of the typed node and typed edge nature of provenance graphs. (CITE) For example, instead of simply counting number of edges from a node, we counted number of edges of type *input* directed towards *file* type nodes, or the number of *fork* edges from *process* type nodes.

Furthermore, we hypothesize that intrusions manifest as anomalies in the lineage of digital artifacts, and one way of observing this is through *graph centrality metrics*. A *graph centrality metric* is a function of a node and its containing graph, and intuitively represents how *central* or *important* the node is in the graph. In terms of provenance graphs, processes of the same program typically behave similarly to one another and this should result in similar number of inputs, outputs, number of ancestors, and number of descendants for normal process nodes of the same program. Intrusions typically change the number of expected inputs and outputs (e.g., a process forking a shell process when it normally does not), and we hypothesize that this can manifest as unusual changes in centrality for the offending node in the provenance graph. ??? note the following 3 observations unique to provenance graphs (CITE):

1. The ubiquity (importance) of a node is a function of a node's descendants, not its ancestors. (Note: Because edges in provenance graphs are in the direction of data dependence instead of data flow, "descendants" in this case means ancestors in the provenance graph).
2. There is no agreed-upon granularity in which provenance should be captured.
3. Provenance has a temporal component.

Using these observations, they propose several centrality metrics on provenance graphs:

- **In-degree Centrality**. Because a node's importance is based on its output descendants, then the number of times a node was used as input to another node indicates a simple measure of importance.

- **Age**. They note that large "jumps" in timestamps or age might indicate breaks between tasks, but this interpretation is complicated when scripts execute tasks in short succession.
- **Ancestor Centrality**. This measures the total number of descendants of a node, normalized by the total number of nodes in a graph.
- **Opsahl's Closeness Centrality**. This accounts for the number of edges between a node and its descendants. To compute this, we calculate

$$CC'(v) = \sum_{x \in V - \{v\}} (d'_{xv})^{-1}$$

where V is the set of nodes in the graph, d'_{xv} is the distance from node x to v by only following outgoing edges from x. Unreachable nodes from x will have $d'_{xv} = \infty$, and so $(d'_{xv})^{-1} = 0$.

- **Provenance Eigenvector Centrality**. The centrality of the nodes is given by the left dominant eigenvector of the matrix:

$$M_{ij} = \begin{cases} 1 & \text{if there is an edge } i \to j \\ 1/|V| & \text{if there is no out edge from } i \\ 0 & \text{otherwise} \end{cases}$$

We implemented some of these graph centrality metrics and used Parzen-Windows on centrality measurements to compute density estimates for each process.

2.4 Intrusion Types

The DARPA Intrusion Detection Data Sets (CITE) describe various kinds of intrusions in a network, including:

1. **Denial of Service (dos)**. Intrusion that results in denial of service, e.g., overwhelming a server.
2. **Remote to Local (r2l)**. Unauthorized access from a machine, e.g., guessing passwords.
3. **User to Root (u2l)**. Unauthorized access to local superuser (root) privileges, e.g., various buffer overflow attacks.
4. **Probing (probe)**. Surveillance and scanning, e.g. port scanning.

Because we only collect provenance graphs local to the machine, we chose to limit our intrusion detection system only to u2l intrusions within this paper, as the other types of attacks involve exploiting the network and would not appear in provenance data.

3 Design and Implementation

3.1 Preliminary Provenance Exploration

To determine which provenance statistics seem useful, we first created simple simulated intrusions and ran them in a Provenance Aware Storage System (PASS) (CITE), a file system with a mounted volume that automatically collects provenance of all files within that volume. To make data collection easier, we used PASSv2, which provides an Ubuntu virtual machine with the PASS volume already installed.

For each of our intrusions, we ran it normally multiple times and collected the provenance. We then ran the intrusion with the exploit multiple times and collected the new provenance.

We used the following procedure to implement our Parzen-Window density estimates and intrusion/normal decision.

1. For each statistic that we implemented, we computed the counts for that statistic for all the nodes in the provenance graph containing the normally-behaving workload.
2. We aggregated the counts by the name of the process (the path to the executable). For a given process P, denote the set of counts for that process name as D_P.
3. For each $y \in D_P$, calculate the log-likelihood $L_P(y)$.
4. For a test node with process name P in the exploited-workload provenance graph, we calculate the value x of the statistic in its graph, and then compute $L_P(x)$. If the ratio of the number of nodes y with $L_P(y) \leq L_P(x)$, to the total number of nodes in D_P is greater than the false detection rate ψ, we label the node as normal. Otherwise we label the node as an intrusion.

In the exploited workloads, we used Orbiter (CITE) to find the abnormally behaving node by hand, and we used these nodes as our test nodes.

3.2 Simulated Intrusions

3.2.1 hello

We found that many of the popular exploits involved buffer overflows and arbitrary code execution and possibly obtain root access. We created a simple `hello` program that echos standard in to standard out, and we gave it a buffer overflow vulnerability that allows a malicious user to fork a shell.

We ran `hello` normally 40 times on various inputs and collected the provenance times. We then exploited the buffer overflow vulnerability in `hello` to spawn an `ls` process, and saved the standard out of that process to a file.

3.2.2 mcrypt

The mcrypt encryption package had a long standing buffer overflow, which would allow a malicious user to run their own code when mcrypt tried to decrypt the crafted exploit file. Due to limitations in our Virtual Machine set up, we were unable to exploit the vulnerable version of mcrypt, much like the authors of Backtracker [3] found. To emulate this exploit, we designed a Python program, which would decrypt (via ROT13) an input file. When sent a particular exploit file which contained malicious lines, our program would stop decrypting and start running the code specified in the file.

We ran our program on over 200 good inputs, and stored the provenance data. We then ran our program on two bad inputs, first to call `ls` as a Proof of Concept, and again to call `/bin/sh` to demonstrate that more severe attacks can be performed. We again exported the provenance generated from this exploited workload.

3.2.3 gcc

Another type of exploit is more reminiscent of a system administration failure. By unexpectedly aliasing an important command, a user can unknowingly grant root access to a malicious program. To emulate this sort of attack, we replaced `/usr/bin/gcc` with our own version of gcc, which, in addition to compiling the user's code (via a call to gcc), logs that the user has made a request and copies the files the user had tried to compile.

For our system administration attack, we compiled the PASS toolset using GCC, and exported the provenance from these runs. We then replaced GCC with our exploited version, compiled the toolset several times with different configurations, and exported the provenance again.

3.3 Selecting Provenance Statistics

Below is the list of simple provenance graph statistics we implemented:

1. number of input and output files
2. number of input and output processes
3. number of input and output pipes
4. number of versions stemming from this node

We also implemented the indegree, Opsahl, and ancestor centrality metrics. We chose not to implement the Provenance Eigenvector Centrality because computing the eigenvectors of the large matrix took too much computation time on our machines. We also chose not to implement age because age would make our data too dependent on how quickly we executed our workloads.

In Table 1, we show a summary of the statistics that had any statistical difference between normal and abnormally behaving nodes. The buffer overflows showed the most abnormal behavior in the number of outgoing processes and the centrality measures. This makes sense, as arbitrary code execution will generate additional processes. Of the centrality metrics we implemented, the Opsahl Centrality metric did the best in distinguishing an exploited node from normally behaving nodes. Because the Opsahl Centrality metric accounts for the number of edges between a node and all of its descendants, this centrality metric is very sensitive to the addition of nodes. Exploits that have additional outputs (e.g. new processes and output files through a remote shell), thus generate a higher Opsahl centrality metric for the offending node. The more active the exploit is (say, if it forks a user shell), the more the centrality measure will be impacted, and the more likely it is that the exploit will be caught by our model.

If the exploit does directly not generate additional files, however, statistics like "number of output files" will fail to be statistically significant. Indeed, the two programs (hello and mcrypt) that did not generate additional files were not flagged by that measure. On the other hand, the node representing the exploited version of `gcc` had abnormal density estimates, as it generated many more files than a normal `gcc` process.

Because of this exploratory analysis, we chose to implement Opsahl centralty and Ancestory centrality and analyze its accuracy on a real intrusion detection data set.

4 Evaluation

4.1 Experimental Setup

talk about SPADE and building graphs from BSM

statistic	hello	mcrypt	gcc
# forked processes	0.0 — 1 — 0.108	0.0 — 1 — 0.107	0.9 — 0 — 0.408
# input files	0.5 — 0 — 0.453	0.5 — 0 — 0.453	0.5 — 1 — 0.107
# output files	0.5 — 1 — 0.453	0.5 — 1 — 0.453	0.0 — 3 — 0.000
Opsahl centrality	1.791 — 2.83 — 0.252	1.250 — 4.867 — 0.000	2.793 — 7.415 — 0.024
Ancestor centrality	0.0009 — 0.0011 — 0.799	0.0002 — 0.0014 — 0.798	0.0007 — 0.0002 — 0.798

Table 1: average over good — test node value — density estimate of test node being "normal"

4.2 Results

4.3 Discussion

5 Future Work

5.1 N-distance Statistics

Some processes fork helper processes that do the real work (such as `gcc`). We found that `gcc` forks processes for the linker/assembler that handles generating output, so `gcc` does not directly have output files. Similarly, when running a python script, the code file itself is not directly executed, it is read in by a python process, which then goes off and collects input, generates output, and forks additional processes. Thus, when running a python file, the file itself only has one node in the provenance graph, with many python process nodes accessing it. In order to account for these kinds of relationships, we could calculate the relevant statistics for the test node and all nodes at distance N from the test node, and then aggregate them. By utilizing information about neighboring nodes, we hope to detect intrusions that involve more complicated relationships between nodes.

5.2 ENV and ARGV

Currently we ignore the environment and argument information stored in the provenance graph. By generating histograms based on this information, we could catch subtler intrusions that rely on modified environment information. Additionally, if an exploit relies on a particular argument or argument patterns, this information might reflected in the ARGV information.

5.3 Online Intrusion Detection

Currently, this system runs after-the-fact, but truly useful intrusion detection systems need to run while the system is online. To do this, we would need to modify our statistics to use centrality metrics that are *progressive*, i.e., they can be recomputed online without needing to recompute the metric on the entire graph.

Given progressive forms of these statistics, we can modify PASS to compute Parzen-Window density estimates as it generates provenance, or we can audit system calls like SPADE (CITE) to build provenance graphs in user space and apply our Parzen-Window technique.

5.4 Local vs. Global

In our experiments, we looked at provenance graphs in their entirety. However, provenance graphs are meant to be long-lived, and an intrusion that affected only a small portion of the graph will have negligible effects on the global structure of the graph. To resolve this issue, a possible future extension to our work is to cluster the nodes based on time [6] and then only examine the most "recent" nodes when applying our histogram technique. Another approach would be to discard provenance data in computing Parzen-Window density estimates after the data have reached a certain age.

6 Conclusion

7 Acknowledgements

We want to thank Margo Seltzer and Daniel Margo for their guidance on the direction of the project and for their help in working with PASS and analyzing provenance graphs. We also want to thank the members of the CS261 Program Committee for their feedback on the initial draft of this paper.

References

[1] Cao, D., Qiu, M., Chen, Z., Hu, F., Zhu, Y., and Wang, B. Intelligent Fuzzy Anomaly Detection of Malicious Software. In *Internal Journal of Advanced Intelligence*, vol. 4, no. 1, pp 69-86 (December 2012).

[2] Inoue, H. and Somayaji, A. Lookahead Pairs and Full Sequences: A Tale of Two Anomaly Detection Methods. In *2nd Annual Symposium on Information Assurance* (June 2007).

[3] King, S. T. and Chen, P. M. Backtracking Intrusions. In *SOSP'03 Proceedings of the nineteenth ACM symposium on Operating systems principles* (December 2003).

[4] King, S. T., Mao Z. M., Lucchetti, D. G., and Chen, P. M. Enriching intrusion alerts through multi-host causality. In *Proceedings of the 2005 Network and Distributed System Security Symposium* (February 2005).

[5] Lei, H. and Duchamp, D. An Analytical Approach to File Prefetching. In *Proceedings of the USENIX 1997 Annual Technical Conference* (January 1997).

[6] Macko, P., Margo, D., Seltzer, M. Local Clustering in Provenance Graphs (Extended Version). In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management* (August 2013).

[7] Macko, P. and Seltzer, M. Provenance Map Orbiter: Interactive Exploration of Large Provenance Graphs. In *TaPP'11 Proceedings of the 2nd conference on Theory and practice of provenance* (June 2011).

[8] Margo, D., and Smogor, R. Using Provenance to Extract Semantic File Attributes. In *TaPP'10 Proceedings of the 2nd conference on Theory and practice of provenance* (February 2010).

[9] Muniswamy-Reddy, K., Braun, U., Holland, D. A., Macko, P., Maclean, D., Margo, D., Seltzer, M., and Smogor, R. Layering in Provenance Systems. In *Proceedings of the 2009 USENIX Annual Technical Conference* (June 2009).

[10] Muniswamy-Reddy, K., Holland, D. A., Braun, U., and Seltzer, M. Provenance-Aware Storage Systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (June 2006).

[11] Offensive Security, Inc. The Exploit Database. `http://www.exploit-db.com`.

[12] Rapid 7 Inc. Metasploit Framework. `http://www.metasploit.com`.

[13] Somayaji, A. and Forrest, S. Automated Response Using System-Call Delays. In *Proceedings of the 2000 USENIX Annual Technical Conference* (August 2000).

[14] Tariq, D., Baig, B., Gehani, A., Mahmood, S., Tahir, R., Aqil, A., and Zaffar, F. Identifying the provenance of correlated anomalies. In *SAC'11 Proceedings of the 2011 ACM Symposium on Applied Computing* (March 2011).