Intrusion Detection through Provenance-Based Histograms

Kenny Yu
Harvard University
kennyyu@college.harvard.edu

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

CS261, Fall 2013

Abstract

Data-driven intrusion detection is a common system security technique. Here we present a method for generating a statistical model based approach to intrusion detection, using provenance data generated by a provenance-aware modified file system. Our analyses reveal that provenance data can be directly used to detect intrusions, and an in-depth analysis reveals which data is most useful in distinguishing between safe and malicious programs and program executions. Our histogram based approach requires no prior intuition about the types of exploits potentially being run on a system, and identifies suspicious behavior based on deviations from prior normal system behavior.

1 Introduction

Provenance is metadata that tracks the history of all changes to files. In provenance-aware storage systems like PASS [10] and PASSv2 [9], the file system automatically tracks all dependencies whenever a file is created, modified, or deleted. These dependencies include the command that was executed to modify the file, the environment of the command, and all input files. The generated provenance forms a directed acyclic graph of typed nodes (e.g. files, processes, pipes) with properties (e.g. name, execution time, pid) and typed edges (e.g. forked, input, versioning).

To make sense of the large amounts of data, Macko et. al. have developed Orbiter, a tool to visualize provenance graphs with semantic zoom [7]. Margo et. al. have developed techniques to analyze large provenance graphs and to extract useful information from these graphs, including semantic file attributes [8]. Using simple provenance statistics on nodes (e.g. neighbors, file count, process count, edge count) and features from `stat`, they determine with high accuracy the type of files (e.g. source files, configuration files). Furthermore, they have identified key properties of provenance graphs and developed new centrality metrics to perform local clustering of graphs, separating the huge provenance graphs into smaller semantic tasks or workloads [6].

Applications that require provenance collection typically place great emphasis on data integrity. Given this priority, a natural goal of provenance systems is to detect intrusions on the system. Somayaji et. al. have developed techniques for detecting intrusions based on system call sequences, and counteract these by exponentially delaying or aborting system calls, rendering the system useless for a malicious attacker [11][2]. These intrusions include exploiting vulnerabilities in the SSH daemon and sendmail to obtain a shell with root privileges. Somayaji's work relies on building a *normal* profile of a process, and then detecting when the process deviates from normal.

Given the extensive amount of data PASS collects on file-file, file-process, and process-process dependencies, it seems plausible to detect intrusions based on provenance data collected by PASS. Tariq et. al. generalize Somayaji's sequences of system calls to sequences of provenance system events [12] to develop an intrusion detection system (IDS) in a distributed system. Their system uses bloom filters to store hashes of k-tuples–representing a sequence of provenance events–and then use these bit vectors to correlate anomalies across multiple hosts. King et. al. have developed Backtracker [3], a modified Linux kernel that tracks dependencies between operating system objects (files, processes, file names) in a similar fashion to PASS. Given an intrusion, Backtracker builds a backwards casual graph to determine the entry point of an intrusion, and they have extended Backtracker with forward causal graphs to determine possibly tainted files, processes, and hosts from an intrusion in a distributed system [4]. However, their system does not find a way of detecting an intrusion.

Lei et. al. build similar process-file dependency trees, which they call access trees, and define a compatibility function to compare access trees with the goal of predicting future file accesses based on the current access tree pattern [5].

Despite the existing work to detect intrusions and anomalies, Cao et. al. argue that any intrusion detection system cannot always distinguish good behaviors from bad behaviors because users behave too randomly to have an accurate model of the user's behavior [1]. They propose their fuzzy anomaly detection approach to extend the training models in existing intrusion detection systems to account for the inherent fuzzy nature of intrusion detection. When training an IDS, instead of assigning hard 0s or 1s for features, their approach provides a way to reliably map these features to weights in the interval $[0, 1]$.

In this paper, we present a technique to analyze existing provenance data to detect if an intrusion occurred.

In particular, we generate statistical models of normal behavior, against which we can test a node of interest. These statiscal models are histograms of different graph statistics, and we use these histograms to generate kernel density estimations to evaluate a test node. The contributions of this paper are:

1. A histogram-based technique for using provenance data to detect intrustions

2. An analysis of the different intrusions that can be identified, and the types of provenance which best contribute to these detections

3. A cogent plan for future work for using provenance data to detect intrusions

2 Background and Related Works

2.1 PASS

Provenance data is meta-data detailing the creation, transformation, and flow of data. PASS [10, 9] is a provenance-aware storage system, designed to generate and track this provenance data seamlessly. The PASS system automatically creates and updates provenance data fore every file that was created or modified the volume. The PASS provenance data is very rich data. For example, the provenance for a data file that sent in a zip file would reveal both the name of the program that unzipped the file, as well as the web browser through which the zip file was downloaded! It is our belief that this level of specificity will lend itself to detecting patterns in program usage.

The PASS system runs with minimal time overhead, and a reasonable space overhead (XXX cite real numbers).

2.2 Intrusion Detection

Intrusion detection in live systems is a common problem in the systems community. There are two primary components to an intrusion detection system (IDS): (1) the data used to fuel the system's analysis, and (2) the process in which this data is used.

Somayaji et al. [11] aimed to detect intrusions by keeping track of all systems calls that occur during a process's execution time. After collecting this data over a long period of "normal" time, if a proces exhibits system call patterns that deviate from the "normal" patterns for the process, their system flags it as potentialy dangerous, and slows its execution. In this way, a program that continually deviates from normal behavior will be descheduled entirely.

Tariq et al. [12] extend the work of Somayaji and others, designing a system to run an IDS on a set of distributed nodes. They use the IDS of other researchers, distilling a programs execution into a set of values (a "k-tuple") used to identify information of interest regarding the program's exeuction. Our system is designed to run offline, so we run our algorithms on the full set of provenance data. While this approach is slower, it allows us to take in the full picture of what occurred during a process's execution, without having to preselect the information we think will be useful. This approach is especially helpful because different information leads to the detection of different types of intrusions.

Backtracker [3] is a system most similar to ours in design. It, like PASS, collects data while the system is running, and like our system analyzes it offline for potential intrusions. Whereas PASS collects data directly at the operating system level (using a modified file system), Backtracker runs its target OS inside of a virtual machine, and uses a modified virtual machine monitor to collect data. The data they collect is very similar to the data PASS generates, and their exploits and analysis techniques directly inspired our work.

3 Design and Implementation

Here we present our intrusion detection technique: a general and extendable statistical method for analyzing and comparing provenance data. At a high

level, we process a provenance data graph, generating a model of "normal" behavior for each process/file in the graph. Using this model, we can determine the likelihood that a given "test" node is normal or whether it deviates from normal behavior and is thus potentially an exploited or malicious program.

3.1 Models

Our approach requires statistical models to evaluate the normalcy of a test node. The statistical model we chose to use was a kernel density estimation (KDE), which estimates the probability density function of a value with unknown distribution. To create the KDE, we needed to provide counts of a particular statistic. In this way, for each statistic that we chose to collect, we would be able to evaluate a test node's normalcy according to that statistic. For example, we could then ask "Does node X_n have a statistically believable number of output processes compared to the other X nodes?"

3.2 Provenance Statistics

As described above, in order to generate a KDE model, we need some statistic to measure. As we expected intrusions to differ in terms of the statistics that would stand out, we needed to generate a large number of statistics, and a model for each. With regards to provenance nodes directly, we compiled the following statistics:

1. The number of input files to the node.
2. The number of input processes to the node.
3. The number of input pipes to the node.
4. The number of output files to the node.
5. The number of output processes to the node.
6. The number of ouput pipes to the node.
7. file_in_version The "version" number of an incoming file.
8. file_out_version The "version" number of an outgoing file.

For ease of computation and comparison, we treated the different node types (e.g. file, process, pipe) separately so that we actually had different counts for "input files to a file node" and "input files to a process node", for instance.

3.3 Centrality Metrics

In addition to the data that can be collected on the nodes directly, we could also compute clustering statistics by considering each node in the context of the larger graph. Using the methods espoused by Macko et al. [6], we could compute different centrality and clustering metrics for a node, which can then be tabulated into a histogram like the input/output counts above. KENNY TODO: EXPAND THIS

3.4 Implementation

The implementation of the above techniques consists of multiple Python files, spanning about 850 lines of (commented) code. The raw BerkeleyDB database generated by PASS is provided as input, and textual/graphical results are outputted. KENNY TODO: EXPAND THIS

3.5 Examples

In Figure 1, we see a few examples of a generated histogram and KDE model. The green bars represent the histogram for the particular statistic, and the blue curve is the KDE. Finally, the red line is the value for a "test" node. In this case, the test node appears to be a possible exploit, because it intersects the KDE at low values.

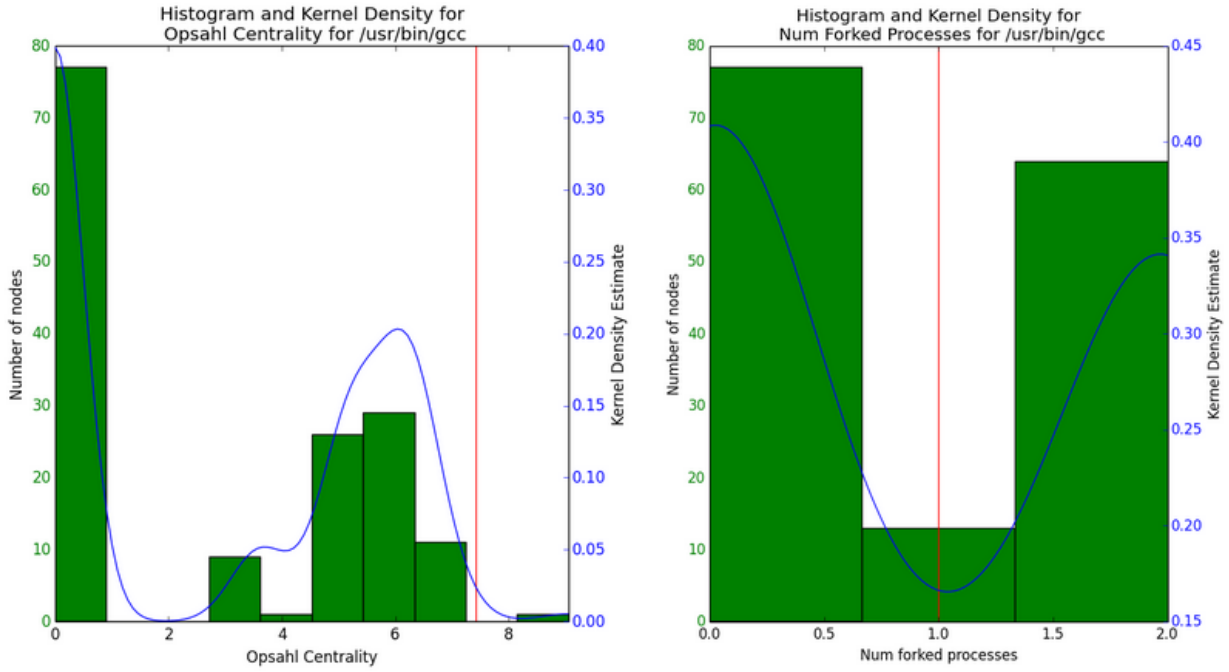
4 Evaluation

4.1 Evaluation Methodology

In order to test our techniques, we needed to run exploits on our PASS-enabled machine. We looked to prior work to get a sense of the different types of techniques commonly used to exploit security vulnerabilities. Commonly, remote network techniques (as seen in Backtracker [3]) are used to broach insecure systems. Metasploit [13] is a software package designed to facilitate running exploits on insecure system. Searching the Metasploit database, as well as the online Exploit Database [14], we found that most successful techniques involve exploiting vulnerabilities in software that a target system is running, as opposed to vulnerabilities in the target system itself. Put another way, we sought to exploit software packages, like mcrypt and gcc, not the Linux kernel itself.

We found that many of the popular exploits involved buffer overflows and arbitrary code execution. Programming bugs in a piece of software allowed a malicious user, sending well-crafted data, to run arbitrary code on the target system. In particular, the

Figure 1: Example histograms of a centrality measure and process statistic, with the assoicated KDE



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 our own 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.

Another type of exploit is more reminiscent of a system administration failure. If a malicious user replaces a program in /usr/bin, it could be used to log user actions. To emulate this sort of attack, we replaced /usr/bin/gcc with our own version of gcc, which, in addition to compiling the users code (via a call to gcc), logs that the user has made a request and copies the files the user had tried to compile.

To generate the provenance data necessary to run the above analyses, we needed to run both good and exploited versions of our programs. For the mcrypt example, 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 by these steps, for comparison with our good usage. This is similar to recording all uses of a program during a servers lifetime.

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, and compiled a few files, and exported the provenance again. In this way, we again acquired provenance for normal usage, before acquiring provenance for our attempted exploit.

4.2 Selecting Metrics

Because we had no prior intuition into what the provenance data would actually look like, we approached selecting metrics from an objective, data-based standpoint. We wanted to use metrics that had variation over our workload, so as not to minimize the introduction of workload dependent data. Towards this end, we graphed all of the metrics over the "normal" provenance data, to see where there were trends and variation. Our results for GCC are in Figures 2 and 3. We can see that proc_out_proc (outgoing processes) and the ancestor and opsahl centrality metrics have more variation, and will be less likely to introduce biases. As we include more data (as discussed in Future Works), these sorts of graphs will become

Figure 2: Histograms over all "simple" provenance statistics for GCC

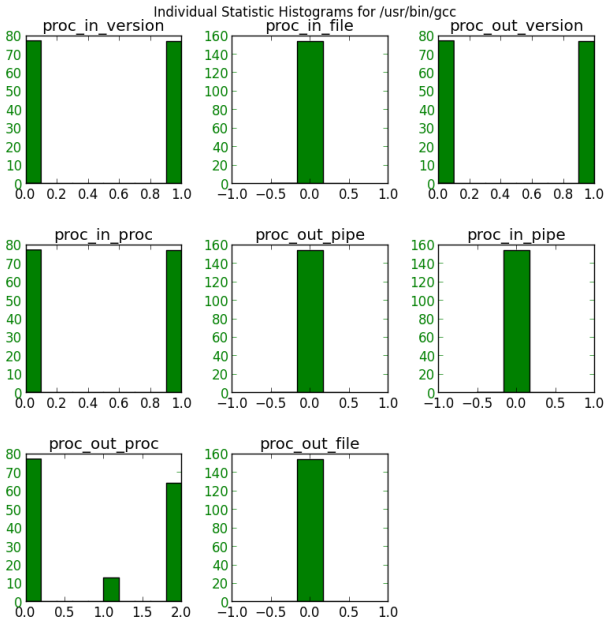
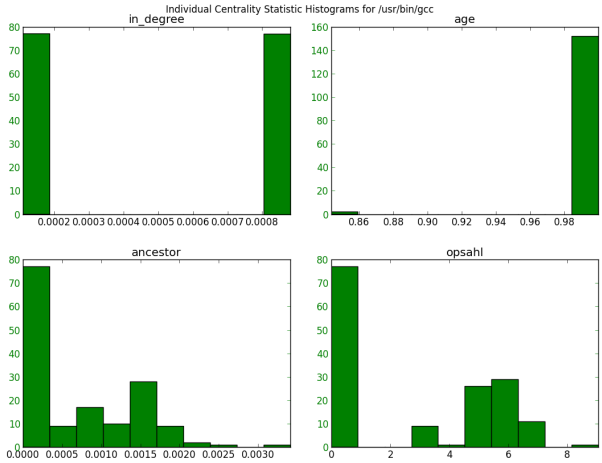


Figure 3: Histograms over all centrality metrics for GCC



chosen four metrics that provided a statistically significant result.

increasingly important in selecting good metrics for evaluation.

4.3 Results

4.4 Provenance Data

After running the normal and exploited programs described above, we visualized the provenance graphs using Orbiter [7]. After searching for the nodes named after our programs, we were able to find both the clean and exploited nodes. The clean nodes for "hello" simply read in one file through a pipe and outputted one file, whereas the exploited version clearly has a forked "/usr/bin/ls" process. Similarly, the mcrypt simulation (implemented in Python) has only an output file as a descendent normally, but when exploited has multiple extra processes/files as descendents (see Figure 4). Similarly, the normal GCC and exploited GCC displayed very different provenance graphs.

4.5 Statistical Data and KDE Predictions

Table 1 summarizes some of the preliminary results we have generated. Here we have included, for each statistic and test, the average over the "normal" nodes, the observed value for the "test" node, and the KDE probability output for that value. We have

4.6 Discussion

4.7 Successes

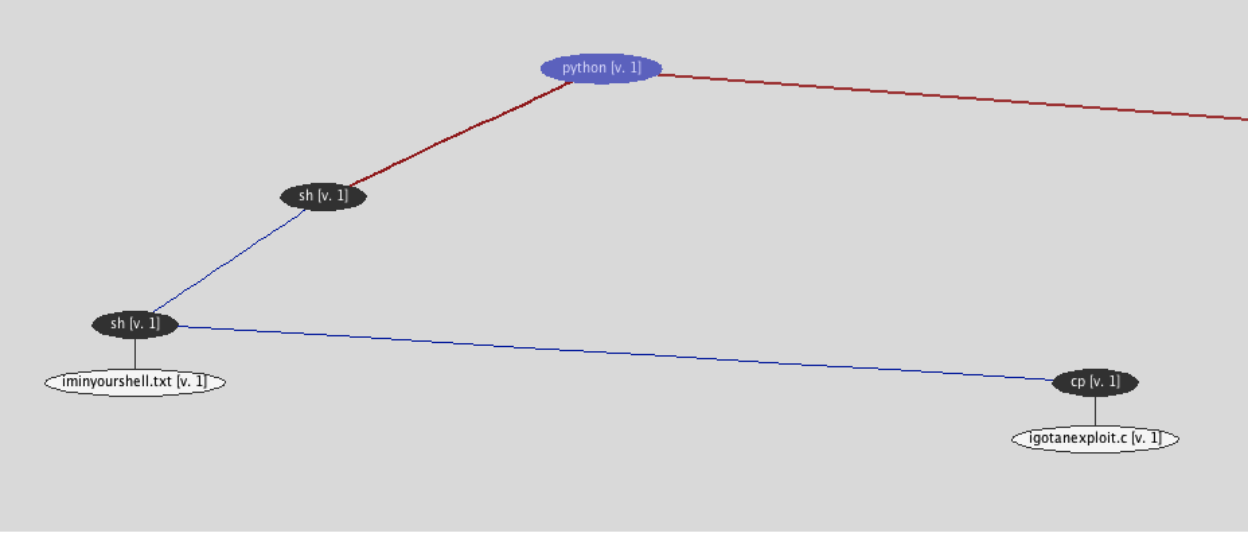
We were able to accurately identify our three exploited programs. Because each was a slightly different exploit, the data on which they deviated differed, but each did have metrics which flagged them as non-normal.

4.8 Useful Data for Intrusion Detection

We set out to identify both buffer overflows (i.e. arbitrary code execution) and modified binaries. The buffer overflows were best detected by the number of outgoing processes and the centrality measures. This makes sense, as arbitrary code execution will generate additional processes. The centrality measures are also very sensitive to the addition of nodes, and additional processes add both a new process node, as well as new nodes for the data touched by the new process. 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.

Figure 4: Descendents of an unexploited mcrypt-style program. Simply an output file. Below, descendents of an exploited mcrypt-style program. In addition to an (offscreen) output file, there is also an outgoing /bin/sh process, with its own children.



metric	hello	mcrypt	gcc
# forked processes	0.0 — 1 — 0.108	0.0 — 1 — 0.107	0.9 — 9 — 0.408
# input files	0.5 — 0 — 0.453	0.5 — 0 — 0.453	0.5 — 1 — 0.107
# ouptut 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

Table 1: average over good — test node value — kde(test node is normal)

On the other hand, the GCC that was modified to generate log files was quickly caught by this measure, as it generated many more files than a normal GCC process.

4.9 Confusing Data Patterns

At first, some of the data we collected seemed wrong. Why, for instance, does GCC seem to normally output zero files? Upon further inspection, 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. To run our analyses, we needed to manually identify this connection, and compute statistics for the python process, instead of the single code file node.

5 Future Work

5.1 N-depth Statistics

Currently, the statistics and counts we generate apply only to the test node itself. As mentioned above, some processes fork helper processes that take in input or generate output, which would be useful information to collect. In order to account for these sorts of actions, we could calculate the relevant statistics for the test node and all nodes at distance N from the test node, and then aggregate them. We could experiment with aggregation techniques to determine how best to incorporate this information. Perhaps it is best to simply sum all neighbor statistics together, or perhaps this information calls for a separate histogram and model. By utilizing information about neighboring nodes, we would hope to see an increase in accuracy and the ability to detect less blatant intrusions.

Additionally, by using this approach, we could avoid the manual process identification mentioned above. On python files, for instance, N-depth statistics would compute statistics for the python process, which is always a direct neighbor of the code file we initially cared about.

5.2 Histograms by Name

Currently, our system compute statistics based on node type, but not node name. For instance, When we compute the outgoing (i.e. "forked") processes, we simply count them. Thus, if a program normally forks something benign, like say "cp", but an exploited version forks "rm", the raw counts may not reveal this (a call to "/bin/sh", however, should impact the centrality measures). One direction to attempt to rectify this problem lies in generating more specific histograms. Instead of having a model for "number of processes forked", we could generate one model per type of process forked, e.g. "number of 'cp' processes forked". This more specific data might enable us to catch intrusions in larger programs that already modify a lot of files and fork a lot of processes.

5.3 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 intrustions/exploits that rely on modified environment information. Additionally, if an exploit relies on particular argument or argument patterns, this information might reflected in the ARGV information.

5.4 Parameters

Currently, constructing the kernel density estimates requires a few parameter decisions - namely the number/size of the bins for our histograms, as well as the bandwidth parameter for construction the KDE. We, for the purposes of this experiment, modified these parameters manually until they described the data well, but we could computer optimal values programmatically, or test out multiple values. Additionally, we demonstrated that odd behavior has a "low" KDE, but we did not define what "low" means. For

this system to be deployed more widely, we will need to define or compute what thresholds signify a potential intrusion or exploit.

5.5 Online Intrusion Detection

Currently, this system runs after-the-fact. We can identify intrusions that occurred on a machine by analyzing the data after it has been lifted from the target machine. Because PASS collect data in real time, however, our approach can be modified to implement online intrusion detection. To acheive this goal, we would need to only use metrics that can be updated without recomputing the entire statistic. In particular, some of the centrality measures do not meet this criterion. We would also need to add additional daemons (or modify the existing PASS daemons) to run the appropriate statistic gathering and model generating code.

6 Conclusion

We have demonstrated that provenance data (in this case, data generated by PASS) can be used to identify anomolies in program behavior, which often signal a security intrusion or exploit. Using provenance-based histograms, we can contruct models of the regular usage patterns for a particular program. When the intrusion or exploit results in behavior deviating from these models, its provenance allows us to flag it as a potential intrusion. Replaced binaries that acted significantly differently were especially easy to detect. Histograms on simple provenance statistics are good for identifying simple anomalies, but more complex statistics (see future work) may enable us to detect subtler intrusions.

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

[12] 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).

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

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