Identifying the Provenance of Correlated Anomalies

Dawood Tariq Basim Baig Ashish Gehani
SRI International*
Menlo Park, CA 94025, USA
{dawood.tariq,basim.baig,ashish.gehani}@sri.com

Salman Mahmood Rashid Tahir Azeem Aqil Fareed Zaffar
Lahore University of Management Sciences
Lahore 54792, Punjab, Pakistan
{salman.mahmood,rashid.tahir,azeem.aqil,fareed.zaffar}@lums.edu.pk

ABSTRACT

Identifying when anomalous activity is correlated in a distributed system is useful for a range of applications from intrusion detection to tracking quality of service. The more specific the logs, the more precise the analysis they allow. However, collecting detailed logs from across a distributed system can deluge the network fabric. We present an architecture that allows fine-grained auditing on individual hosts, space-efficient representation of anomalous activity that can be centrally correlated, and tracing anomalies back to individual files and processes in the system. A key contribution is the design of an *anomaly-provenance bridge* that allows opaque digests of anomalies to be mapped back to their associated provenance.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

anomalies, correlation, distributed, monitoring, detection, Grid, provenance, lineage

1. INTRODUCTION

Grid applications harness the power of a large number of distributed nodes to solve complex problems that cannot be solved on a single machine in a reasonable amount of time. The popular volunteer computing Grid application SETI@home [28] computes 710 TFLOPS while analyzing radio telescope data in search of extraterrestrial intelligence. Modern pathogens on the Internet are capable of causing significant damage. The epidemic nature of the infections they cause limits the time in which security experts can respond and protect their systems. The immense processing power and networking bandwidth afforded to Grid infrastructures and the applications running on them makes the Grid a highly attractive target [27].

Although access to a Grid may be controlled through authentication mechanisms [9], there is little control over security vulnerabilities in the customized applications developed by legitimate users to address their specific research problems [7, 5, 11, 10, 16, 2, 9]. In the event that a running application is compromised by an attacker, a Grid's single-sign-on authentication can be leveraged. Once the trust is breached, all nodes accessible to the compromised application within and outside the Grid are threatened by the once-trusted node. Compromised nodes can then be used for a variety of attacks, ranging from the deployment of botnets or spam relays to the storage of pirated data.

Timely and credible security information is therefore critical to adaptive network and application security management. Monitoring Grid applications requires fine-grained auditing of applications at the nodes. The resulting log files can then be analyzed centrally to determine anomalous activity. Several such intrusion detection and Grid application monitoring systems [14, 17, 19] have been proposed in the past. Since fine-grained auditing results in a deluge of data, a compromise is typically reached by logging events at coarse granularity [24]. The resulting loss of information reduces the ability to isolate the data that has been compromised once an attack has been detected.

We present the design and implementation of an architecture that relies on a fine-grained audit log along with a data provenance collection system. Storing the provenance of all the files and processes is essential for isolating the causes and effects of an attack. A space-efficient data structure termed the *anomaly-provenance bridge* (APB) is used to support mapping from anomalous events in the audit log to specific provenance elements. Using the APB with an intrusion detection system facilitates the generation of augmented threat digests that can be used for offline taint analysis of the files accessed and processes executed during an attack.

The event monitoring architecture is designed for use with Grid applications and focuses on minimizing the storage and

*This material is based upon work supported by the National Science Foundation under Grant OCI-0722068. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

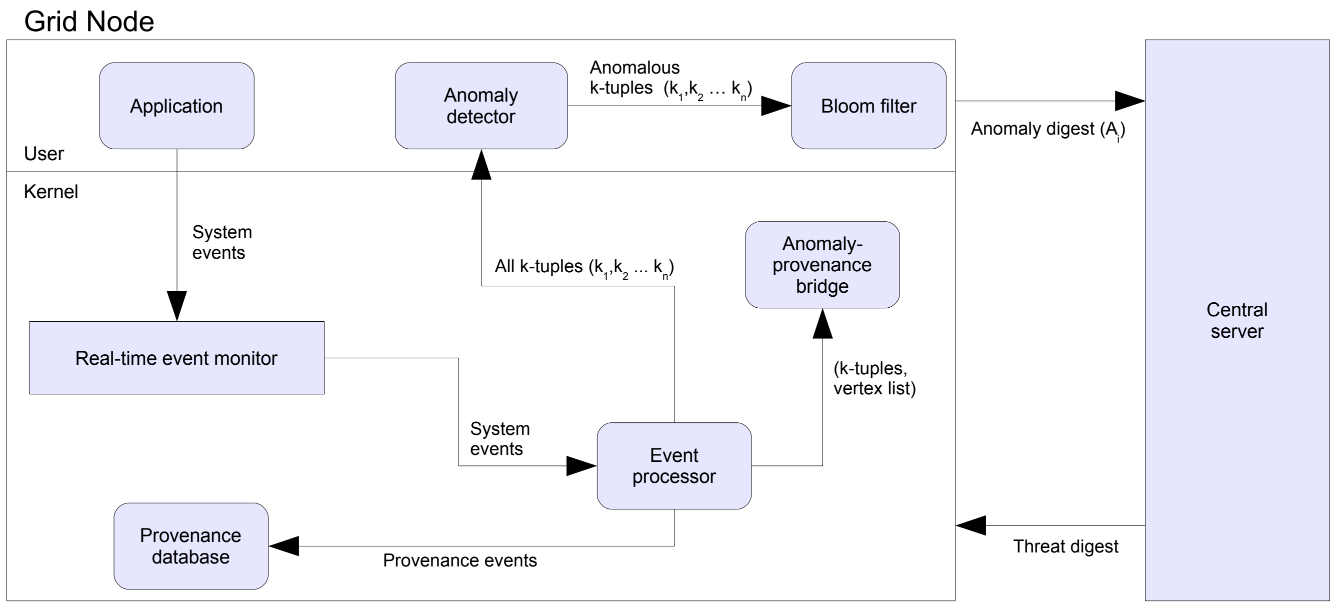

Figure 1: After system events are processed, entries in the provenance database and the anomaly-provenance bridge are created, and anomalous activity is sent to a central server.

network bandwidth overhead. It is employed while demonstrating the efficacy of the APB under actual attack conditions. Although a community of nodes running instances of BOINC [1] is used as an exemplar in this paper, the architecture and hence the results are applicable in the context of other Grid applications.

Section 2 describes how anomalous activity is correlated across the distributed system. Section 3 explains the design and use of the APB. Section 4 outlines the experiments performed. Related work is described in Section 5 before we conclude in Section 6.

2. CORRELATING ANOMALIES

Our goal is to correlate reports of anomalous activity coming from different hosts in a distributed system and then be able to trace the sources. To achieve this, each node logs system activity, summarizes it, and sends it to a central server where correlation is performed. The key challenge we address is how to simultaneously provide sufficiently detailed information to the server for correlation to be effective while limiting the network bandwidth and host storage requirements. The architecture of the system is depicted in Figure 1.

2.1 Host Monitoring

Each host being monitored in the distributed system is instrumented to audit all system events along with their arguments. The interaction between applications and the kernel on a host is characterized by the stream of system events invoked. A sequence of k system events is referred to as a k-tuple, and is the basis of analysis of a number of anomaly detection schemes, such as that of Forrest [14], Malan [19], and Oliner [24]. While our framework supports the implementation of such schemes by reporting k-tuples observed on monitored hosts to the central server, we do not introduce any new detection algorithms.

2.2 Training

Anomaly detection requires a training phase during which usual system activity is monitored and recorded, with the resulting profile used to characterize *normal* behavior. To support this, the set of all k-tuples observed during this period is logged. We have previously observed that this log grows rapidly [12] but can be effectively represented using a Bloom filter [3]. The i^{th} host H_i will have a Bloom filter B_i to which all k-tuples $k_1, k_2, \ldots$ that are observed are added:

$$H_i \quad : \quad B_i \quad \leftarrow \quad \{\ k_1, k_2, \ldots\ \}$$

2.3 Local Anomalies

During normal operation at host H_i, if a k-tuple $k_j \notin B_i$ is observed that had not been seen during training, it is anomalous by definition and must be sent to the central server. In practice, the stream of k-tuples generated is too voluminous to send in raw form to the central server. Oliner [24] reported that with 35 clients being monitored, the server received up to 20 Mb/s. Sending the raw stream for hundreds of Grid hosts could deluge the server performing correlation and generate substantial network traffic. Instead, hosts collect observed anomalies for a period of time that we term an *epoch*. In the t^{th} epoch, the i^{th} host H_i will construct an *anomaly digest* $A_i(t)$:

$$H_i \quad : \quad A_i(t) \quad \leftarrow \quad \{\ k_1 \notin B_i, k_2 \notin B_i, \ldots\ \}$$

In the same way that the profile of normal k-tuples was implemented with a Bloom filter, anomaly digests are also created using a Bloom filter to save storage on individual hosts and network bandwidth when transmitting them over the network. At the end of epoch t, the anomaly digest $A_i(t)$ is sent to the server S:

$$H_i \quad \rightarrow \quad S \quad : \quad A_i(t)$$

2.4 Calculating Correlation

At the end of epoch t, the server calculates a correlation digest $D(t)$ using the anomaly digests it has received. (An extension could use digests from past epochs to incorporate a memory of anomalous activity.) What constitutes correlation between sequences seen on distributed hosts depends on the specific anomaly detection algorithm. We describe two general examples of calculating correlation using anomaly digests.

Static Thresholding

If the same k-tuple has been observed on more than a threshold number T_c hosts within an epoch, then that k-tuple is deemed to be correlated and should be included in the correlation digest. The server can calculate this efficiently by first constructing a counting filter [8] $C(t)$ with the j^{th} bucket being the count of the number of anomaly digests received in epoch t with their j^{th} bit set, and then setting every bit in $D(t)$ to 1 if the corresponding bucket in $C(t)$ is over T_c and otherwise to 0:

$$S \quad : \quad C(t)[j] \quad = \quad \sum_i A_i(t)[j]$$

$$S \quad : \quad D(t)[j] \quad = \quad \begin{cases} 1 & : \quad C(t)[j] \; > \; T_c \\ 0 & : \quad C(t)[j] \; \leq \; T_c \end{cases}$$

Dynamic Thresholding

If the rate at which anomalous activity is occurring does not vary dramatically, static thresholding works well. An alternative approach is to vary the threshold as a function of the anomalous activity. As more hosts generate more anomalies, the server's threshold should increase, and vice versa. This requires the server to estimate the activity level on hosts in order to adapt the threshold T_c. This can be done efficiently by using counting filters instead of Bloom filters for the anomaly digests $A_i(t)$, and then normalizing a baseline threshold T_b by the sum of the buckets of all the anomaly digests:

$$S \quad : \quad T_c \quad = \quad \frac{T_b}{\sum_i \sum_j A_i(t)[j]}$$

The correlation digest is then calculated using the same algorithm that is used for static thresholds.

2.5 Updating Hosts

After the server has calculated a new correlation digest $D(t)$, it sends the digest to every host H_i in the system:

$$S \quad \rightarrow \quad H_i \quad : \quad D(t)$$

The digest provides each host with a view of anomalous activity that is occurring across the system. In particular, it can be used as a *threat digest* [12] with intrusive activity on more than T_c hosts automatically resulting in a digest at all the remaining hosts that allows the same activity to be recognized and flagged before the attack succeeds.

3. ANOMALY-PROVENANCE BRIDGE

When a host identifies an anomalous k-tuple, we are interested in determining which process was responsible for generating the k-tuple as well as which files had been tainted by the process. Recording system events along with all their arguments would allow the relevant processes and files to be identified but create a very large audit log. Instead, we leverage the fact that most arguments are not relevant. We create a data provenance subsystem, and a bridge that allows a specific k-tuple to be mapped to its provenance, as depicted in Figure 2.

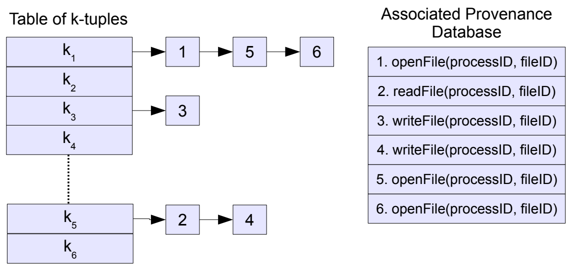
Table of k-tuples
k1
k2
k3
k4
k5
k6
1
5
6
3
2
4
Associated Provenance Database
1. openFile(processID, fileID)
2. readFile(processID, fileID)
3. writeFile(processID, fileID)
4. writeFile(processID, fileID)
5. openFile(processID, fileID)
6. openFile(processID, fileID)

Figure 2: The *anomaly-provenance bridge* is a space-efficient data structure that links anomalous sequences to entries in the provenance database. For example, looking up the k-tuple k_5 yields identifiers 2 and 4, corresponding to vertices in the database.

3.1 Data Provenance

The provenance of a data object is characterized by the set of processes that have modified it and recursively the provenance of all the data objects that were inputs for those processes. More details are accessible in the specification of the Open Provenance Model [25]. Each host in the system runs its own provenance collection service, which creates a process vertex for each program that is run, a file vertex for each new version of a file, and read and write edges connecting process and file vertices.

3.2 Indexing k-tuples

Each time an anomalous k-tuple k_j is observed on host H_i, the arguments of all k system events are inspected. The set of provenance vertices $V_j = \{v_1, v_2, \ldots\}$, corresponding to the process and files that are arguments of the j^{th} k-tuple of system events, is constructed and added to any existing set in the APB:

$$H_i \;\; : \;\; APB_i(k_j) \;\; \leftarrow \;\; \begin{cases} V_j & : \;\; APB_i(k_j) = \emptyset \\ V_j \cup APB_i(k_j) & : \;\; APB_i(k_j) \neq \emptyset \end{cases}$$

Every provenance-related system event will manifest in k sequential k-tuples, necessitating a maximum of k new entries in the APB. Since many sequences repeat in practice, very few new entries are needed for an average k-tuple. Instead, provenance vertices may need to be added.

3.3 Utilization

Consider the case where a vulnerable distributed application is running on a Grid being monitored by our system.

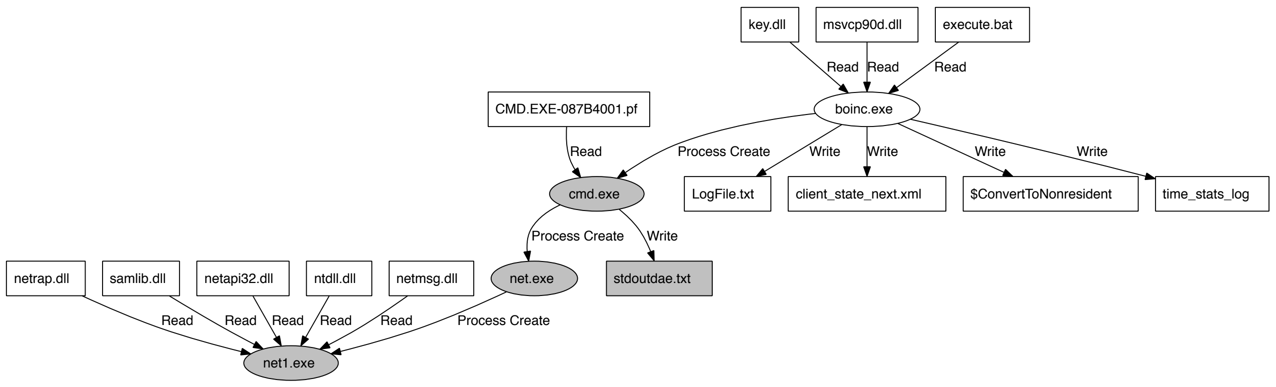

Figure 3: A provenance graph highlighting the flow of a synthetic DLL injection attack on a host running an instance of BOINC. The attack created a user account as a backdoor on the host machine.

If malware such as a worm or bot exploited the application, anomalies would begin to be generated at affected hosts. These would be correlated by the server, which would then generate threat digests and disseminate them to all the hosts in the system. The threat digest can be used in conjunction with the APB to identify current and past compromises.

Real-time Response

If a k-tuple of system events occurs on a host and the sequence is present in the threat digest, the APB can be inspected to see if there is a corresponding entry. If so, the host infection has been detected in real time and the k-tuple can be used to map to a set of provenance vertices. By tracing back through the provenance graph, tainted data and compromised processes can be identified.

Behavioral Integrity

The second case occurs if a k-tuple in the threat digest had manifested on the host before it was flagged by the central server. In such a case it would not have been handled in real time. However, if the behavioral integrity of the host needs to be verified, every k-tuple in the APB can be checked against the threat digest. If any APB entries match, the host has been infected in the past. As in the case of real-time response, the k-tuple can then be mapped to provenance vertices using the APB.

4. EVALUATION

4.1 Platform

Our experiments were performed on Microsoft Windows XP (Service Pack 3) running on a 2.8 GHz Intel Core2Duo processor with 3.5 GB of memory. Auditing was effected with Process Monitor 2.91 [20]. Bloom filters were constructed with the Open Bloom Filter [23] library. Provenance records were stored in MySQL Community Server 5.1.48 [21].

4.2 Workload

We ran the volunteer computing application BOINC 6.10.56 to construct a profile of normal behavior. Intrusions were synthetically created using Windows' *CreateRemoteThread()* to inject DLLs into BOINC's address space. Four scenarios were constructed. The first scenario simulated insertion of a spam relay by inserting Mailboy 2004 [18] with its internal SMTP and DNS servers. Mailboy sent email to 1,700 addresses using 20 threads with a timeout of 30 seconds between messages. The second scenario mimicked a stealth mode relay, effecting the same functionality at a much slower rate. The third case simulated a multimodal attack with one of three propagation vectors selected randomly. The three choices were insertion of a spam relay, installation of a rootkit using an FTP exploit, and creation of a user account as a backdoor. (The associated provenance graph is depicted in Figure 3.) The fourth attack was the same as the third but with slower propagation.

4.3 Performance

Recall that the APB allows a k-tuple to be mapped to a set of provenance vertices associated with the arguments of the system events in the k-tuple. Figure 4 plots the storage needed to record every k-tuple of system events and the associated arguments. Over a 25 hour monitoring period, the space required for the raw log grows to 155.9 MB. Using the APB, the storage needed drops to 18.6 MB since multiple occurrences of the same k-tuple result in a single APB record.

During the normal operation of a host, new k-tuples will be observed and added to the APB. In addition, when new instances of the same k-tuple are observed the arguments are likely to differ and so the set of vertices associated with the k-tuple grows as well. Since we store the sets as linked lists, we refer to the cardinality of the vertex set as the *chain length* associated with a k-tuple.

When the provenance of a k-tuple is checked using the APB, the set of vertices associated with *all* past instances is returned. Some of these vertices may have been associated with a different occurrence of the k-tuple under consideration. Vertices of this latter subset are *false provenance reports* and their presence increases as the chain length grows. To address this, we prune the chains periodically. In our experiments, the pruning occurred after two hours.

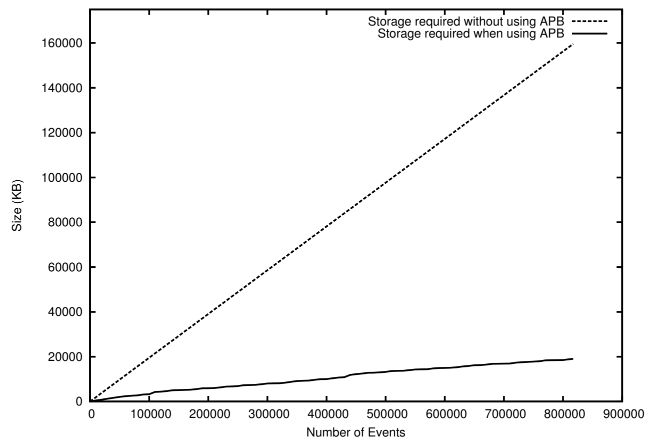

Figure 4: The amount of storage utilized with and without the anomaly-provenance bridge for mapping k-tuples to system event arguments.

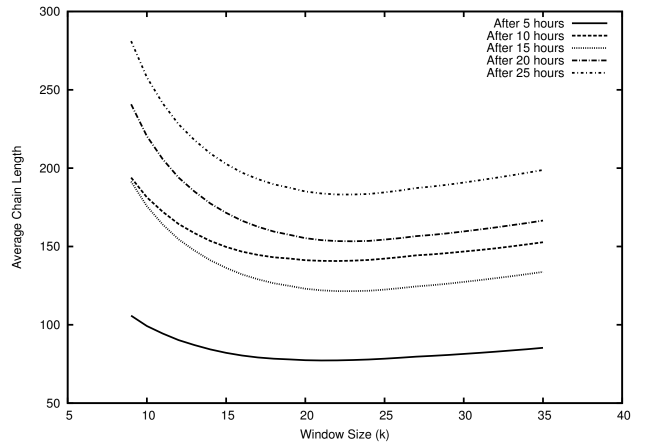

Figure 5: As the size of the k-tuples used grows, the average chain length first drops and then starts increasing. This is of interest since the chain length corresponds to the false positive likelihood of a vertex being reported in the provenance of the k-tuple.

Since longer chains result in more false provenance reports, we investigated how they could be minimized. When monitoring the stream of system events, we used a sliding window of size k to generate the k-tuples. As k was increased, we noticed that the false provenance reports reduced. This occurs because a larger value of k results in more unique k-tuples, decreasing the likelihood that two different application execution sequences result in the same k-tuple of system events.

As we continued increasing k, we observed that the average chain length started increasing after a point. Figure 5 shows that the inflection occurred when k crossed 23. This dynamic occurs because many anomalous k-tuples are already present in the APB after an initial period. When the same k-tuple is observed repeatedly, more provenance vertices are added to the list associated with it, increasing the chain length.

Increasing the sliding window size also has an adverse impact on the amount of storage needed since the number of unique sequences grows exponentially with k. This creates an incentive to minimize the value of k used. However, smaller values of k result in reduced discrimination between anomalous and benign behavior, in addition to increased false provenance reports.

Since the choice of k is a tradeoff, we would like to find a value that minimizes the average chain length. This value may vary over time depending on how quickly the chains associated with different k-tuples grow. We therefore studied the effect of increasing the workload duration, as shown in Figure 5.

We expected the average chain length observed for a given value of k to always increase (since more provenance vertices are associated with each k-tuple over time). Initially, the chain lengths did grow. However, after 15 hours the average chain length was lower than it was 5 hours earlier, for all values of k. Subsequently, the average chain length started growing again. The intermediate decrease in average chain length occurred because the rate at which new provenance vertices were being created was dominated by the growth rate of the number of new anomalous k-tuples being generated.

Of particular note is the fact that the value of k that minimizes the average chain length remains independent of the size of the workload.

5. RELATED WORK

Behavior-based application monitoring needs a means to differentiate between normal and abnormal activity of the application. One method for defining *normal* was presented by Forrest et al. [14] The group defined a *sense of self* for UNIX processes by recording consecutive sequences of system calls during a training period and storing the sequences in a database. Any sequence of system calls that is not in the database is declared anomalous. The number of normal sequences falsely flagged as anomalous can be reduced by increasing the period of training at the cost of failing to identify more anomalous activity as such. Keromytis's group argued that the homogeneity of *application communities* could be used to decrease false positives without increasing false negatives [17]. Oliner's work on temporally correlating anomaly signals extends this. In contrast, Malan and Smith [19] demonstrated that user behavior is uncorrelated and strong temporal correlation is indicative of the presence of worms and bots.

Software-based taint analysis techniques [22] typically require applications to be instrumented with the concomitant runtime overhead limiting scalability of the approach. One strategy to reduce the overhead is to focus on particularly vulnerable parts of the system, such as Web browser plugins. An example of this is the dynamic classification of Browser Helper Objects (BHOs) into those that generate malicious versus benign activity. Data that is tainted by malicious activity can then be rolled back [6]. Eudaemon [26] generalizes the approach and allows any Linux application to be monitored by redirecting execution to a software emulator that is augmented to perform taint analysis. Since emulation imposes a high overhead, a virtualization-based effort has been explored to reduce the overhead enough to track the entire system [13]. Finally, the Minos system [4] explores the use of the Biba integrity model to track the integrity of control flow data, incorporating the taint analysis into the hardware layer.

Though application behavior monitoring and taint analysis have both been studied in detail, there has been relatively little effort focused on bridging the two domains [15], because the efforts have been undertaken in the context of a single host, with records from one subsystem available to the other. In contrast, our work is in the distributed setting of a Grid where communication costs are substantial, precluding reliance on a global view of the detailed logs.

6. CONCLUSION

The anomaly-provenance bridge enables detailed analysis of the provenance of anomalous activity. This analysis can be used to trace infections to their respective source files and processes with acceptable false positives and minimal storage overhead. A better understanding of the pathogen via its provenance record allows for effective and timely immunization in order to thwart the epidemic spread of threats. The security and privacy concerns that typically inhibit forensic data sharing are minimized by the use of summaries. We validated the approach using synthetically created attack scenarios with the BOINC volunteer Grid computing platform and DLL injection to introduce malicious code. Of particular interest was the relationship between the size of each k-tuple and the number of provenance artifacts associated with it, which dropped as k grew to 23 and then started increasing again.

7. REFERENCES

[1] David P. Anderson, BOINC: A system for public-resource computing and storage, 5th IEEE/ACM International Workshop on Grid Computing, 2004.

[2] Asia Pacific Grid Policy Management Authority, `http://www.apgridpma.org`

[3] Burton H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM, Vol. 13(7), 1970.

[4] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong, Minos: Architectural support for protecting control data, ACM Transactions on Architecture and Code Optimization, 2006.

[5] European Data Grid, `http://cern.ch/edg`

[6] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song, Dynamic spyware analysis, USENIX Annual Technical Conference, 2007.

[7] European Grid Policy Management Authority, `http://www.eugridpma.org`

[8] Li Fan, Pei Cao, and Jussara Almeida, Summary Cache: A scalable wide-area Web cache sharing protocol, IEEE/ACM Transactions on Networking, Vol. 8(3), 2000.

[9] Fermilab Public-Key Infrastructure, `http://computing.fnal.gov/security/pki/`

[10] I. Foster, C. Kesselman, and S. Tuecke, The anatomy of the Grid: Enabling scalable virtual organizations, International Journal of High performance Computing Applications, 2001.

[11] Globus, `http://www.globus.org`

[12] Ashish Gehani, Basim Baig, Salman Mahmood, Dawood Tariq, and Fareed Zaffar, Fine-Grained Tracking of Grid Infections, 11th ACM/IEEE International Conference on Grid Computing, 2010.

[13] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand, Practical taint-based protection using demand emulation, 1st ACM European Conference on Computer Systems, 2006.

[14] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji, Intrusion using sequences of system calls, Journal of Computer Security, 1998.

[15] Xuxian Jiang, Aaron Walters, Florian Buchholz, Dongyan Xu, Yi-min Wang, and Eugene H. Spafford, Provenance-aware tracing of worm break-in and contaminations: A process coloring approach, 26th IEEE International Conference on Distributed Computing Systems, 2006.

[16] LCG Security Group, `http://cern.ch/proj-lcg-security`

[17] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis, Application communities: Using monoculture for dependability, 1st Workshop on Hot Topics in System Dependability, 2005.

[18] Mailboy 2004, `http://massmailersoft.com/index.asp`

[19] David J. Malan and Michael D. Smith, Exploiting temporal consistency to reduce false positives in host-based collaborative detection of worms, 4th ACM Workshop on Recurring Malcode, 2006.

[20] Process Monitor 2.91, `http://technet.microsoft.com/en\_us/sysinternals/bb896645.aspx`

[21] MySQL Community Server, `http://dev.mysql.com/downloads/mysql/`

[22] James Newsome and Dawn Xiaodong Song, Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, 12th Network and Distributed System Security Symposium, 2005.

[23] Open Bloom Filter, `http://www.partow.net/downloads/OpenBloomFilter.zip`

[24] Adam J. Oliner, Ashutosh Kulkarni, and Alexander Aiken, Community epidemic detection using time-correlated anomalies, 13th International Symposium on Recent Advances in Intrusion Detection, 2010.

[25] Open Provenance Model, `http://www.openprovenance.org/`

[26] Georgios Portokalidis and Herbert Bos, Eudaemon: Involuntary and on-demand emulation against zero-day exploits, 3rd ACM European Conference on Computer Systems, 2008.

[27] Symantec Corporation, Internet Security Threat Report: Vol. XV, April 2010.

[28] D. Werthimer, J. Cobb, M. Lebofsky, D. Anderson, and E. Korpela, SETI@home - massively distributed computing for SETI, IEEE Computing in Science and Engineering, 2001.