Intrusion Detection Using N-Grams of Object Access Graph Components

Zachary Birnbaum Andrey Dolgikh Victor Skormin
Binghamton University,
Binghamton, NY, USA
{zbirnba1, adolgik1, vskormin}@binghamton.edu

Abstract - **Cyber warfare demonstrates an arms race between mutually escalating malware and Intrusion Detection System (IDS) technologies. We put forward a novel process for defining system behavior with the end result being a highly effective IDS. System calls accumulated under normal network operation are converted into graph components, and used as part of the IDS normalcy profile. This paper are as follows: detection of attacks based on the anomalous use of program functionality; reduced window of attack; reduced false positive rate; increased performance in comparison to standard n-gram methods; a graph compression algorithm for efficient processing of system call graphs. The proposed IDS can be used within limited access environments such as industrial or military systems where only approved applications are running and any anomalies are indicative of a cyber attack or malfunction.**

Keywords: security, intrusion detection, behavioral anomaly detection, graph processing

I. INTRODUCTION

Modern computer systems, especially those employed in industrial control and automation spheres, usually feature a diverse software stack and unique configuration. This peculiarity was mentioned as the "Diversity hypothesis" in [1]. The diverse environment results in unique computer system operation as seen from the system call level. This can be successfully used to develop a customized behavioral profile tailored to the particular system under consideration. Customized profiling allows an anomaly detection approach to be used. By comparing the previously established profile with the profile of the running system any extracurricular activity within the system in question can be detected and flagged as an anomaly. Hofmeyr et al in [2] shows that intrusions and certain abnormal situations (e.g. lack of disk space or client misconfiguration) trigger anomaly detections. Therefore the detection of anomalous activity may alert a system operator of an intrusion or abnormal system operation.

II. RELATED WORK

A large number of approaches were developed and studied for anomaly based intrusion detection. We will limit our review to system call based Intrusion Detection Systems (IDS).

Forrest et al in [3] offers a simple and effective method based on the n-gram model:

A sequence of n elements of the same type is called n-gram. For example a trigram consists of three elements (w_1, w_2, w_3). The elements can be of any nature such as numbers, words of natural language, or system calls.

The n-gram model operates as follows:

1. The string of elementary observations $S = w_1, w_2, w_3, \dots, w_k$ over the alphabet of possible observations $\sum$ is transformed into a string of n-grams using a sliding window of size n. For example, for $n = 3$ we will get the following string of trigrams:
$$S^3 = (w_1, w_2, w_3), (w_2, w_3, w_4), (w_3, w_4, w_5), \dots, (w_{k-2}, w_{k-1}, w_k)$$
2. Learning phase: observed n-grams S_i^n are accumulated into the database D:
$$D = \bigcup_i S_i^n$$
3. Detection phase: each observed n-gram w is tested if it belongs to database D. If $w \notin D$ (w does not belong to D) the anomaly is detected.

Since its introduction, many modifications of the n-gram model have been offered, but with marginal improvement [4, 5, 6]. Recent notable large scale efforts to apply the n-gram model to malware detection were carried out by Lanzi et al in [7]. Lanzi performed large scale data collection covering ten different hosts under normal use over a prolonged period of time and thousands of malware samples. Forrest and Lanzi built their n-gram IDS with an observations alphabet $\sum$ equal to the set of system calls defined by the monitored system. Both models normally result in a high rate of anomaly/malicious n-gram detections. Therefore, some mechanism was needed to separate real attack n-grams from false detection. The Forrest approach uses the fact that an attack usually occurs within a very short time period and generates bursts of anomalous n-grams. To average out false positives and highlight the attacks, gram-to-gram Hamming distance and normalization over the trace length was used. Lanzi used the detection count of malicious n-grams exposed by each program. When the detection count crossed the threshold value an attack was declared. In spite of these techniques both approaches still have a high false positive rate that prevents their successful application in practice. This can be attributed to the inability of n-grams to distinguish long range dependencies from noise.

Methods different in nature from n-gram models use various kinds of additional information to recover and monitor program control or data flow [8, 9, 10, 11, 12].

Our approach exceeds the performance of the simple n-gram model by using recognized data flow graph components as a source alphabet. Graph components capture completed and semantically meaningful sequences of system calls. The use of graph components helps to eliminate the mentioned inability of n-gram models to capture long contexts. Thus our approach combines n-gram and data flow approaches to cover long spans of program operation.

II.1. Our Approach

The principal idea behind our behavior based IDS is the detection of anomalous use of known program functionalities. In other words, the IDS detects non-standard, previously unseen use of known program functions.

In order to establish a behavioral profile of the program our IDS consumes intercepted system calls with their respective pa-

Copyright (c) IARIA, 2014.

rameters similar to Kolbitsch [13] and Mutz [10]. Using this information, we can trace how each OS object is accessed. This access history can be represented by an Object Access Graph (OAG). After a sufficiently long time, the OAG represents the essence of normal system operation. This facilitates structural anomaly detection, i.e. the detection of anomalous graph components not seen before in the system.

To capture the context of different program functionalities or structural components we use n-gram model. At this point anomalies can be detected by comparing the n-grams obtained from the current OAG to n-grams accumulated during the learning stage.

Any OAG component of a running system can be incorporated into the normalcy graph, thus rendering reinforcement of the structural normalcy profile trivial. Moreover, detected anomalous components labeled by a human expert as malicious can be instantly added to the malicious profile and later recognized as malware skipping the last n-gram detection stage.

II.2. Contributions

We report the following contributions:

- Detection of attacks from the anomalous use of program functionality (section 3.8).
- Reduced false positive rate in comparison to standard n-gram methods (section 4.4.a).
- Reduced window of attack (section 4.4.b).
- Increased performance in comparison to standard n-gram methods (section 4.5).

III. SYSTEM OVERVIEW

III.1. System Call Monitoring

A number of methods exist to intercept system calls and extract their parameters on the majority of OSs. Kernel driver enabled techniques demonstrate negligible overhead [9, 11]. For convenient research and testing purposes, we chose the Linux kernel and the *strace* system call monitoring program as our platform [14]. The research conducted on this platform is generic and can easily be duplicated on other platforms given they provide similar data.

Linux provides approximately 200 different system calls. *Strace*, a debugging utility, is included in the Linux operating system and is capable of monitoring system calls from all non-system processes [14]. To support a system wide monitoring approach, we use *strace* options that allow us to capture data from processes created after system call monitoring began.

III.2. Data Parsing

Strace output contains very useful information, including the time of the system call, the system call name, and most importantly, the argument values for the system call. The argument values vary depending on the system call, but all relevant information will be listed in the *strace* output. A typical output of *strace*:

```
PID   Time  Syscall   Parameters
4734    1    open     ("test.txt")= 8
4734    2    dup2     (8, 5)          = 5
3668    3    open     ("test2.txt")=15
4734    4    close    (8)    = 0
4734    5    write    (5, {"R"}, X) = 0
3668    6    close    (15)            = 0
```

Figure 1. Sample of *strace* output

As seen in the sample of strace output, the process 4734 at time 1 opened the file "test.txt" and has a handle of 8. Any subsequent calls using the same object refer to handle 8 instead of the specific file "test.txt." This parameter value dependency is a key concept that is crucial to building an Object Access Graph.

III.3. Object Access Graph

Observing system calls with their parameters provides a useful model of system behavior. This model is represented by a vertex-edge, directed, and acyclic graph constructed from the parsed *strace* output and can be described as follows:

$$G_m=(V, E, F_v, F_e)$$

where
V – set of vertices,
E – set of edges,
F_v - mapping from V to set of system calls S.
F_e - mapping from E to set of system call argument types T.

The graph G_m can be built from the *strace* data according to the following rules:

- Labeled vertex v_s is added to G_m for each issued system call s.
- Labeled edge e_τ from v_i to v_j is added for system calls i and j when one of the parameters of i and j have the same data type τ, have equal data value d, and have one of the following:
  - v_i has d as the output and v_j takes d as the input
  - v_i was the last system call registered before v_j

For example, *open* and *dup2* (Figure 1) occur at times 1 and 2, respectively, and have a common parameter of handle type equal to 8. In the resulting graph (Figure 2b), nodes corresponding to calls *open* and *dup2* are connected with the directed edge 8. Nodes *dup2* and *close* are also connected with an edge labeled 8 because the *close* that occurred at time 4 uses the same handle 8.

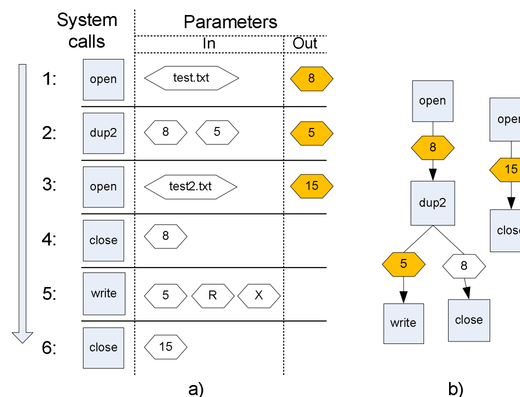

Figure 2. Conversion of system call stream (a) into Object Access Graph (b)

These rules allow us to trace how each OS object is used by different processes. Unlike program centric approaches taken by the majority of Behavior Based IDS [8, 13], this method centers on the system wide behavioral picture imposed by programs over OS objects.

III.4. Graph Component Detection

There are certain terminating system calls defined by the kernel (e.g. *close*, *exit_group*). Once these system calls have been executed over a particular OS object reference, there can be no additional system calls using this reference. In the OAG this means that the component cannot be extended once all its leaves end with terminating calls. Consider the OAG in Figure 3 which contains three completed components.

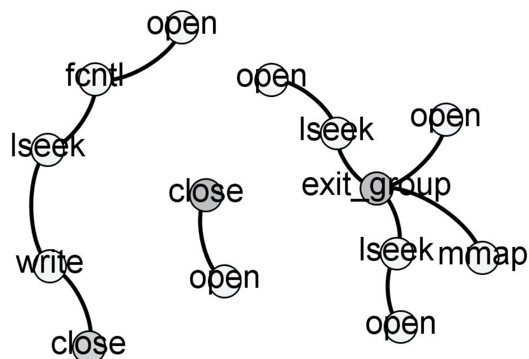

Figure 3. Completed components in OAG

Component detection transforms the stream of system calls into a stream of completed OAG components.

III.5. Component Compression

Due to the repetitive/cyclic actions usually performed by programs over OS objects, some OAG components may grow to unmanageable sizes. However, repetitive occurrences of a single system call or some graph substructures do not provide the observer with substantial additional information. Moreover, it has a detrimental effect on graph recognition.

Consider the graphs featured in Figure 4.a, 4.c. These graph instances represent typical large system call graph components. These graphs are simple in nature but have large node counts that reflect repetitive operations usually performed by programs over one or two OS objects. For example, a network input-output routine may repeatedly send data in small chunks, generating long chains of sends/receives over the socket handle. This type of behavior impairs the recognition process in two ways: First, large graphs take a lot of computing resources to process them. Second, the number of substructure repetitions and consequently the graph component size may depend on factors irrelevant to exposed behavior. This in turn leads to unnecessary component duplication in the database of known components. Therefore it is beneficial to remove/collapse repetitive subgraphs as shown in Figure 4.b, 4.d. It reduces processing load and substantially decreases the number of different graphs observed throughout program operation.

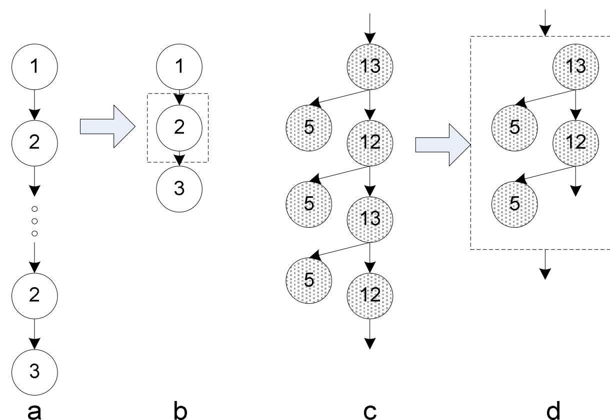

Figure 4. Graph compression

We perform frequent subgraph compression in two stages: First we remove long repetitive chains of single call as show in Figure 4 (a, b). Second, we apply the modified Graphitour algorithm [15] to find and remove/collapse more complex repetitive components.

III.6. Graph Component Database

After compression every completed graph component *c* is subjected to the following normalcy profiling algorithm:

```
 Input: completed component c,
        set of components DB={d1, d2, …}
 Output: Component database DB
 ----------------------------
 Begin
1   foreach d ∈ DB = {d_1, d_2, … d_n} do
2      if IsIsomorphic(c, d) then continue
3      else Db = Db ∪ c
 End
```

Figure 5. Normalcy profiling algorithm

Where the function *IsIsomorphic* tests if two graphs have the same structure.

The algorithm produces a compact database containing one instance of each observed graph component. Using this database, we can detect components that have not been previously encountered. Therefore, it constitutes a normalcy profile of the system.

III.7. Anomalous Component Detection

Once malware has been introduced to the system, it will perform its mission, resulting in additional system calls. Malware functionality will differ from normal system operation and new, unknown OAG components will be observable.

```
Input: completed component c,
        set of components DB={d1, d2, …}
 Output: Match or No_match
 ----------------------------
 Begin
1   foreach d ∈ DB = {d_1, d_2, … d_n} do
2      if IsIsomorphic(c, d) then return Match
4   return No_match
 End
```

Figure 6. Anomaly detection algorithm

These new components, along with components consistent with standard operation, are fed to the anomaly detection algorithm. The anomaly detection algorithm is similar to the profiling algorithm in Figure 5. However, it returns No_match instead of updating the database, when an unknown component is detected.

At this point, we are able to detect the manifestation of malware intrusion in the domain of system call graph components. Using this approach, malware becomes discernible system-wide at a higher semantic level.

III.8. N-grams applied to graph components

To detect the anomalous use of known program functionalities identified by the algorithm in Figure 6 we apply the n-gram model. The model operates as follows:

1. The string of system call observations $S = w_1, w_2, w_3, \dots, w_k$ is converted into string of graph components $S' = c_1, c_2, c_3, \dots, c_k$ using algorithm featured in section 3.3.
2. Sliding window is used to convert string of observed graph components into string of graph-n-grams: $S'^n = (c_1, \dots, c_n), (c_2, \dots, c_{n+1}), \dots, (c_{k-n-1}, \dots, c_k)$.
3. Learning phase: graph-n-grams are accumulated into database $$D = \bigcup_i S'^n_i$$
4. Detection phase: each observed graph-n-gram is tested if it belongs to accumulated database. All graph-n-grams that are not present in normalcy database are detected as anomalous.

Figure 7. Learning algorithm for n-gram components over graphs

N-grams over OAG components capture system behavior at the level of program functional blocks such as complete network IO or file editing. This allows us to detect tampering with program control flow at a higher semantic level.

IV. Experimental Evaluation

In this section we provide experimental evaluation for each step of the detection pipeline. The experimental data is available to the public [22].

IV.1. Experimental Setup

In order to evaluate the IDS, we utilized the experiment featuring three computers connected to a common network: victim computer, attack computer, and IDS computer. The victim computer represents the Metasploitable Virtual Machine [16]. The Metasploitable Virtual Machine is an OS package, preconfigured with many exploitable services. In our experiments, we used FTP server (vsFTPd 2.3.4), Samba service (version 3.0.20-Debian), and HTTP Apache server (version 2.2.8) with PHP (version 5.2.12) installed. The victim computer was running a customized *strace* program, which forwarded the system call stream to IDS computer.

The attacking computer is represented by Backtrack Linux, packaged with the Metasploit framework. Metasploit is a software package which comes with tools for vulnerability scanning and penetration testing [17]. Using Metasploit on the attacking computer, we mounted an exploit against services on the victim machine.

A third computer acts as our Intrusion Detection System. The IDS assembles system calls sent from the monitored victim into OAG components. It then passes components into the anomaly detector. At the same time, all activity at the victim host is visualized for expert analysis.

IV.2. Component Database Stabilization

To confirm that our IDS is capable of extracting a limited size OAG component database by processing a volume of system calls data, we ran several tests with loads of different natures: no load, FTP load, HTTP/PHP load, and Samba load. We exercised the FTP server with two different FTP clients by repeatedly connecting/disconnecting, copying small and large volumes of data, changing file permissions, etc. The same approach was taken with the Samba server. The HTTP/PHP server was tested by manual browsing through the sample web site. The results are presented in Figure 8, where the Y-axis illustrates the total number of components in the OAG component database profile, and the X-axis represents the system runtime in number of system calls. With time, all four graphs quickly flatten out, showing that the normalcy profile converges to certain size, which represents all functionality exercised by system.

For each type of load, the absolute number of learned graph components, its average, and maximum sizes stored within the profile are presented in Table 1.

TABLE 1. NORMALCY PROFILE METRICS

	No Load	Samba	FTP	HTTP/ PHP	Combined Profile
Largest Component	10	8	8	12	12
Number of System Calls	94556	575800	483458	107086	1260900
Number of Nodes in Components	55	42	57	83	130
Number of Components	13	11	15	17	26

A useful feature of our IDS approach is the ability to combine different normalcy profiles into one normalcy profile. The combined profile can be used to recognize behaviors from each incorporated profile. By its nature, the combined profile cannot contain duplicate components, therefore keeping only one copy of each. One may assume that components from the no load profile must be present in all subsequent profiles as a background. This is not entirely correct as background operations performed by various daemons are time dependent. Therefore, certain operations observed in the no load profile are not present in Samba profile as we run Samba profiling at a different time. The combined profile automatically takes care of such issues.

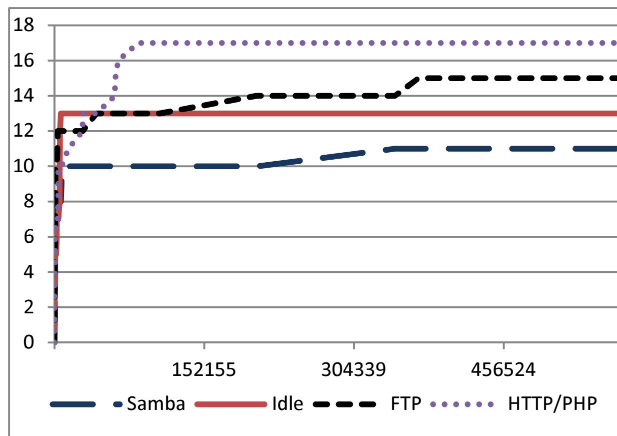

Figure 8. Stabilization of the graph component database size.

IV.3. N-gram Database Stabilization

The next step is to confirm that the database of n-grams learned from the stream of OAG components according to algorithm in Figure 7 converges to a limited size. Figure 9 shows that the database converges for both direct system call n-grams and OAG n-grams. This stabilization was observed under various loads: Idle, Samba, HTTP/PHP, and FTP.

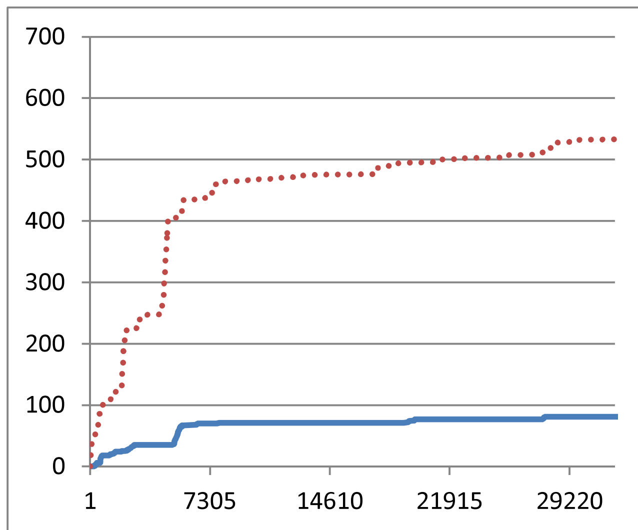

Figure 9. Stabilization of component 3-grams (solid line) and system call 3-grams (dotted line) under no load

One may notice that OAG n-gram database has much lower n-gram counts in comparison to direct system call n-grams. We attribute this to the ability of our system to capture logically finished sequences of program actions.

TABLE 2. COMPARISON OF N-GRAM PROFILE DATA

	# Syscalls	n	OAG n-gram DB size	direct n-grams DB size
No Load	94556	3	91	628
		5	194	1159
		10	390	2510
Samba	575800	3	67	449
		5	125	858
		10	384	2398
FTP	483458	3	182	2808
		5	467	9775
		10	1111	36005
HTTP/ PHP	107086	3	126	1222
		5	263	2517
		10	558	4767

The pitfall of direct n-grams is a small viewing window. As a result, events connected over a longer period are viewed by system as anomalies. The OAG-based system has much wider context awareness. As a result, it makes fewer errors and requires fewer n-grams to achieve similar or better performance. As seen in Table 2, longer n-grams require larger databases to cover the same activity. Regardless of n, the OAG approach requires a significantly smaller database to capture program normal behavior.

IV.4. Anomaly Detection

In this section, we discuss the detection capability of the anomaly detection algorithm under three types of loads: file and print services (Samba), web services (HTTP/PHP), and file transferring services (FTP).

Metasploitable, serving as our victim machine, is prepackaged with vulnerable services (see Table 3).

All experiments were performed in real time according to the procedure featured in Figure 10. For all tests the n-gram normalcy profile is represented by a merged normal behavior for all loads.

```
0. Start the IDS machine and load n-gram database.
1. Start victim host.
2. Enable system call tracing on the victim host.
3. Start vulnerable service on the victim host.
4. Exercise the service with a normal load.
5. Launch the attack against vulnerable server.
6. Observe detected anomalous n-grams.
```

Figure 10. Experiment procedure

In the experiments, we demonstrated the ability of OAG n-gram and direct n-gram approaches to successfully detect anomalies induced by exploitation attacks. Table 4 summarizes empirical data obtained in the experiments.

Both approaches register anomalies induced by performed attacks. Our OAG n-gram method matches the detection performance of the direct n-gram approach.

TABLE 3. EXPERIMENTAL SETUP

Version	Samba	3.020-Debian
	HTTP/PHP	Apache 2.2.8/ 5.2.12
	FTP	vsFTPd 2.3.4
Exploit	Samba	*/multi/samba/usermap_script*
	HTTP/PHP	*/multi/http/php_cgi_arg_injection*
	FTP	*/unix/ftp/vsftpd_234_backdoor*
Normal activity test	Samba	upload, download, delete, and create files and folders
	HTTP/PHP	browsing hosted pages
	FTP	upload, download, delete, and create files and folders

Increasing values of n results in a greater number of anomalous grams detected. Therefore larger values of n are beneficial for increased sensitivity to attacks. Regardless of the gram size OAG approach shows lower anomaly counts. This is due to the OAG approach operating on a higher semantic level (graph component level) than direct n-grams (system call level).

TABLE 4. ANOMALY DETECTION

		Anomalies	
	n	direct n-gram	OAG n-gram
FTP	3	182	27
	5	552	99
	10	1376	252
SAMBA	3	211	24
	5	509	81
	10	1028	261
HTTP/ PHP	3	54	3
	5	136	51
	10	341	82

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-324-7

IV.4.1. False Positive Rate

To measure the false positive rate we repeated the experimental procedure featured in Figure 10 however no attack was launched (no step 5). The results of the experiments are summarized in Table 5.

TABLE 5. FALSE POSITIVE RATE

		False positives	
	n	direct n-gram	OAG n-gram
FTP	3	144	6
	5	319	22
	10	668	60
SAMBA	3	1065	12
	5	2374	41
	10	4520	92
HTTP/PHP	3	49	3
	5	292	16
	10	467	47

The OAG approach produces fewer false positives than the direct n-gram approach across all services.

IV.4.2. False Negative Rate

Wagner and Dean proposed a statistical mimicry attack against n-gram based methods [18]. We significantly reduce the window of opportunity for such attacks. Now the attacker would need to mimic n-gram statistics, OAG components and OAG n-gram statistics to successfully evade detection. Wagner's method relies on generating long lists of dummy system calls. These dummy system call sequences applied randomly will not match the OAG profile. Therefore proper implementation of Wanger's attack under OAG will require generation of system call sequences that match the OAG profile. This means the attacker cannot sneak in any new functionality and is forced to use functional components already present in the system normalcy profile.

IV.5. Performance Evaluation

We present the runtime performance of our IDS in two dimensions: capturing overhead and detection overhead.

Capturing overhead measures the performance penalty incurred by *strace*. Tests performed using Samba, FTP, and PHP with *strace* enabled did not show noticeable slowdown. A synthetic test using a custom program designed to stress system call interface showed a tenfold runtime increase. The capturing overhead is dependent upon the mechanism used. For example, a kernel driver implementation will result in negligible overhead ([9] reports less than 6% overhead).

TABLE 6. DETECTION OVERHEAD

		Trace Length		Time Spend Detecting	
	n	system calls	OAG components	Direct	OAG
FTP	3	11971	346	0.26	0.29
	5	11971	346	1.006	0.29
	10	11971	346	5.25	0.29
Samba	3	55163	341	9.25	0.35
	5	55163	341	34.29	0.37
	10	55163	341	128	0.36
HTTP/PHP	3	9503	586	0.36	0.53
	5	9503	586	2.62	0.56
	10	9503	586	15.55	0.53

Detection overhead measures the anomaly detector performance's impact on system operation. For OAG based approach it includes reduction of raw data into graph components and graph gram matching. Table 6 shows that OAG n-gram matching approach is extremely efficient, resulting lower overhead when compared to the direct n-gram method with larger n. As n increases, the OAG approach doesn't slow down unlike the direct method. This can be attributed to a much smaller database of OAG n-grams.

V. LIMITATIONS

Our approach assumes a tamper-free data source. We do not have the ability to detect attacks completely hidden by rootkits [19]. However if a rootkit is used to hide only a certain subset of system calls it is likely to break the dependence of calls within OAG components or OAG n-grams thus revealing itself as an anomaly. The same is true for attacks relying on race conditions [20].

Attacks that do not change the program control flow (such as [21]) are not detected by our IDS. However, several important cases of such attacks are still detectable. For example, when the attack alters the data flow, it results in changes of the OAG.

Attacks that make use of misconfigured resources in a legitimate way may not be detected if the same functionality is routine.

Our approach for system normalcy relies on past system behavior. Any unidentifiable future behavior, benign or malicious, as previously discussed, will trigger an anomaly.

VI. CONCLUSION

The widespread use of malicious software continues to be an ever-growing concern. Our research resulted in a prototype IDS built on two key ideas: The transformation of system calls into graph components and matching their sequences.

The IDS employs several novel concepts for program data flow processing. We establish an Object Access Graph (OAG) representing interdependent program operations over OS objects. The OAG is compressed to efficiently represent the essence of program activity. OAG components are subjected to a well known n-gram method.

The developed IDS yielded promising results in several aspects. First, the IDS can detect attacks disguised as normal system operation by using existing program functionality. Second, our method significantly reduces the attack window by monitoring program behavior at different semantic levels.

Experiments demonstrated both a reduction in the false positive rate as well as increased performance when compared to standard n-gram methods. Results also showed that the IDS is capable of detecting unknown attacks against system services.

Our results show that achieving efficient anomaly detection is possible through the intelligent application of graph processing algorithms to system behavioral profiling.

ACKNOWLEDGMENTS

This research is funded by the Air Force Office of Scientific Research (AFOSR) project FA9550-12-1-0077. The authors are grateful to Dr. Robert Herklotz for supporting this effort.

References

[1] S. Forrest, S. Hofmeyr, A. Somayaji, “The Evolution of System-Call Monitoring,” Proceedings of the 2008 Annual Computer Security Applications Conference, 2008

[2] S. Hofmeyr, S. Forrest, A. Somayaji, “Intrusion detection using sequences of system calls,” Journal in Computer Security. 6, pp. 151-180, 1998

[3] S. Forrest, S. Hofmeyr, A. Somayaji, T. Longstaff, “A sense of self for Unix processes, ”Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on , vol., no., pp. 120-128, 6-8 May 1996

[4] C. Warrender, S. Forrest; B. Pearlmutter, “Detecting intrusions using system calls: alternative data models,” Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on , vol., no., pp.133-145, 1999

[5] A. B. Somayaji, “Operating System Stability and Security Through Process Homeostasis,” Ph.D. Dissertation. The University of New Mexico. 2002

[6] N. Hubballi, S. Biswas, S. Nandi, “Sequencegram: n-gram modeling of system calls for program based anomaly detection,” Communication Systems and Networks (COMSNETS), 2011 Third International Conference on , vol., no., pp.1-10, 4-8 Jan. 2011

[7] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, E. Kirda, “Access Miner: using system-centric models for malware protection,” Proceedings of the 17th ACM conference on Computer and communications security, pp. 399-412, 2010

[8] R. Sekar, M. Bendre, P. Bollineni, D. Dhurjati, “A fast automaton-based approach for detecting anomalous program behaviors,” IEEE Symposium on Security and Privacy, pp. 141, 2001

[9] D. Gao, M. Reiter, D. Song, “Gray-box extraction of execution graphs for anomaly detection,” Proceedings of the 11th ACM conference on Computer and communications security, pp. 318-329, 2004

[10] D. Mutz, F. Valeur, G. Vigna, C. Kruegel, “Anomalous system call detection,” ACM Trans. Inf. Syst. Secur. 9, 1, pp. 61-93, 2006

[11] A. Tokhtabayev, V. Skormin and A. Dolgikh, “Expressive, Efficient and Obfuscation Resilient Behavior Based IDS,” Proc. European Symposium on Research in Computer Security, pp. 698-715, 2010

[12] V. Zwanger, F. Freiling, “Kernel mode API spectroscopy for incident response and digital forensics,” In Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, article 3, 2013

[13] C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, X.Feng Wang, “Effective and efficient malware detection at the end host,” Proceedings of the 18th conference on USENIX security symposium, pp. 351-366, 2009

[14] Online. strace software, http://linux.die.net/man/1/strace, retrieved Feb 2013

[15] L. Peshkin, “Structure induction by lossless graph compression,” In Proceedings of the 2007 Data Compression Conference (DCC '07, pp. 53-62, 2007

[16] Online. Metasploitable Virtual Machine, https://community.rapid7.com/docs/DOC-1875, retrieved Feb 2013

[17] Online. Metasploit Software, http://www.metasploit.com/, retrieved Feb 2013

[18] D. Wagner, D. Dean, “Intrusion Detection via Static Analysis,” In Proceedings of the 2001 IEEE Symposium on Security and Privacy, pp. 156, (SP '01)

[19] A. Srivastava, A. Lanzi, J. Giffin, D. Balzarotti, “Operating system interface obfuscation and the revealing of hidden operations,” Proceedings of the 8th international conference on Detection of intrusions and malware, and vulnerability assessment, pp. 214-233, 2011

[20] R. Watson, “Exploiting concurrency vulnerabilities in system call wrappers,” Proceedings of the first USENIX workshop on Offensive Technologies, article 2, 2007

[21] C. Parampalli, R. Sekar, R. Johnson, “A practical mimicry attack against powerful system-call monitors,” Proceedings of the 2008 ACM symposium on Information, computer and communications security, pp. 156-167,2008

[22] Online, strace traces, http://testbed.binghamton.edu/traces

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-324-7