Appeared in Proceedings of the 12th IEEE International Multitopic Conference (IEEE INMIC 2008), pages 373–378. IEEE. Dec. 2008.

Evaluation of Control Flow Traces in Software Applications for Intrusion Detection*

Imran Asad Gul, Nils Sommer, Matthias Rohr, André van Hoorn, and Wilhelm Hasselbring
Software Engineering Group, University of Oldenburg, Germany

Abstract—**Software security has become an important requirement, particularly for systems that are publicly accessible through the Internet. Such systems can be equipped with intrusion detection systems to uncover security breaches.**

In this paper, we present a novel application-level intrusion detection approach. A normal behavior profile is created from application-internal control flow in terms of operation execution traces. Anomalous control flows indicative for intrusion attempts are detected by continuously monitoring and analyzing the software system. A case study demonstrates the intrusion detection approach's applicability in the context of a multi-user Java Web application.

Keywords: *Intrusion Detection, Anomaly Detection, Trace analysis, Markov chains*

I. Introduction

More and more financial and business services are provided through the Internet. Since these services are publicly accessible, this makes them attractive targets for cyber attacks. In 2007 it has been surveyed [1] that 46% of 494 companies reported computer security incidents during the past 12 months leading to a total estimated loss of 67 million US $ compared to 52 million US $ in 2006.

Intrusion detection systems (IDS) are a means for increasing security of Internet and network services. An intrusion can be considered a breach of a computer system security policy and IDSs gather information from within the system and the network in order to uncover intrusions [2].

This paper introduces an IDS based on application control flow analysis. First, the IDS learns a normal application behavior profile in terms of a Markov model. Possible intrusions are then detected by comparing this profile with the current application control flow. A case study demonstrates how the IDS detects intrusions into a Web application.

Markov models have been proposed for normal behavior profiles in intrusion detection approaches for a long time (e.g., [3]). The main novelties of our approach are that the control flow in the application layer is analyzed to detect intrusions and that an additional behavior clustering step has been introduced in the construction of the normal behavior profile. The application layer refers to the software that runs on top of general purpose middleware environments such as application servers (see Schmidt [4]). The additional clustering step creates multiple trace-specific Markov chains in contrast to just one more general Markov chain in order to improve the intrusion detection quality.

The remainder of this paper is structured as follows. Section II provides the basic concepts of IDSs. An outline of our approach is given in Section III, followed by a description of the normal application profile creation and the anomaly detection procedure in Sections IV and V. Section VI provides the case study. Related work follows in Section VII before the conclusions are drawn in Section VIII.

II. Background

Intrusion detection approaches can be classified into the following two categories:

1) *Anomaly Detection*. A normal behavior profile is automatically created from monitoring data and any deviating activity indicates a possible intrusion. A benefit of this technique is that the profile is system-specific. However, a common problem is to achieve sufficiently low false alarm rates, as all behavior that was not part of the monitoring data that was used to create the profile is considered anomalous.
2) *Misuse Detection*. Intrusion are discovered by comparing activities to a repository of patterns for known attacks. This approach can reliably detect known intrusions but it usually fails to detect new types of attacks.

Intrusion detection systems are not a replacement for preventive security mechanisms, such as access control and authentication [5]. Moreover, the combination of IDSs and proactive security mechanisms can achieve a better coverage against security threats than one of these strategies alone.

Intrusion detection approaches can be futher categorized based on what types of activities are monitored. A large class of approaches focus on monitoring and analyzing network packets (e.g., [6]), while other approaches analyze user behavior (e.g., [3]), and system behavior (e.g., system calls sequences by UNIX processes [7, 8]).

III. Outline of our Intrusion Detection Approach

As depicted in Figure 1, our IDS has three major components: Application Monitor, Control Flow Analyzer, and Anomaly Detector. The *Application Monitor* records operation executions within the software application under supervision into a repository. The *Control Flow Analyzer* and its plug-ins synthesize the control flow, i.e., the execution sequences, of each user request stored in the repository and generate the normal application behavior profile using Markov chains. This

* This work is supported by the German Research Foundation (DFG), grant GRK 1076/1.

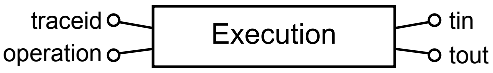


Fig. 2. Data schema of the repository data.

profile is used by the *Anomaly Detector* to detect abnormal behavior in newly incoming monitoring data.

Our intrusion detection approach is implemented as an extension of our monitoring and control flow analysis framework Kieker[1] [9].

We assume that software system under supervision is composed of components. The components provide *operations* that might be requested by other components, external users, or systems. Primary artifacts of runtime behavior are *executions* of the operations. A finite sequence of executions resulting from a request is denoted a *trace*. We limit the scope to synchronous communication between executions as defined in the UML [10]: the caller of an operation is blocked and has to wait until the callee returns a result before it continues its own execution. A trace is a complete representation of the control flow origination from a request.

We represent an execution as a tuple $(traceid, o, tin, tout)$ of an operation o, the start time tin, end time $tout$, and a $traceid$ which is a unique identifier for all executions within the same trace. As mentioned above, traces are ordered sequences of executions. The repository consists of all executions monitored. Figure 2 shows the data schema of the monitoring data and Table I lists an example.

IV. Normal Application Behavior Profile Creation

The creation of the normal application behavior profiles starts by constructing a Markov model from a sufficiently large set of traces that can be considered to be representative for normal behavior. Clustering is used to improve the intrusion detection quality. The approach is detailed in the remainder of this section.

A. Markov Chain Generation

Markov chains provide a common stochastic means to describe dynamic system behavior, for example in reliability and performance engineering. A (first order) Markov chain is a probabilistic finite state machine with a dedicated entry and a dedicated exit state. Each transition is weighted by a probability. For each state, the sum of all outgoing transition probabilities must be 1. Given the current state, the next state is randomly selected solely based on the probabilities associated with the outgoing transition.

The algorithm to transform the monitoring data to Markov chains involves two major steps. First, traces are synthesized from the monitoring data. In a second step, these traces are transformed into Markov chains. In the following the procedure is explained in detail and demonstrated by a running example consisting of the three traces contained in the monitoring data of Table I.

TABLE I
Monitoring data example.

traceid	operation	tin	tout
1	A.a()	0	150
1	C.c()	30	50
1	B.b()	60	140
1	C.c()	90	130
2	A.a()	310	460
2	C.c()	340	358
2	B.b()	370	450
2	C.c()	400	437
3	A.a()	480	535
3	C.c()	510	520

a) From Monitoring Data to Traces: Figure 3 lists the three traces from Table I in a so-called message trace representation [9]. Each message included in such a trace is a tuple of a message type (call or return) and a pair of operation executions, where the first execution is the message sender and the second execution is its receiver. The \$-sign stands for the initial action. Multiple executions of the same operation within a trace are distinguished (e.g., executions C.c() and C.c()$'$ in traces 1 and 2). Message traces can be visualized as UML Sequence Diagrams [10]. The traces 1 and 2 have the same control flow; they only differ in the timing behavior. The resulting UML Sequence Diagrams for the three example traces are displayed in Figure 4.

Trace 1:
(C,\$,A.a()) , (C,A.a(),C.c()) , (R,C.c(),A.a()) , (C,A.a(),B.b()),
(C,B.b(),C.c()$'$) , (R,C.c()$'$,B.b()) , (R,B.b(),A.a()) , (R,A.a(),\$)

Trace 2:
(C,\$,A.a()) , (C,A.a(),C.c()) , (R,C.c(),A.a()) , (C,A.a(),B.b()),
(C,B.b(),C.c()$'$) , (R,C.c()$'$,B.b()) , (R,B.b(),A.a()) , (R,A.a(),\$)

Trace 3:
(C,\$,A.a()) , (C,A.a(),C.c()) , (R,C.c(),A.a()) , (R,A.a(),\$)

Fig. 3. Message trace representation of the monitoring data in Table I.

b) From Traces to Markov Chains: The traces are transformed into Markov chains as follows. At first each message trace is converted into a finite state machine that only accepts this particular trace. Each message corresponds to an individual state and therefore the number of messages is equal to the number of states. The state corresponding to the first message is the state machine's entry state and the state corresponding to the last message in the trace is the end state. The transitions of the state machine are simply based on the order of messages within the trace. Therefore, all states have exactly one incoming transition and one outgoing transition, except the start and end state which have no incoming or no outgoing transaction, respectively. The result of this trace transformation step is a set of finite state machines.

[1] http://kieker.sourceforge.net

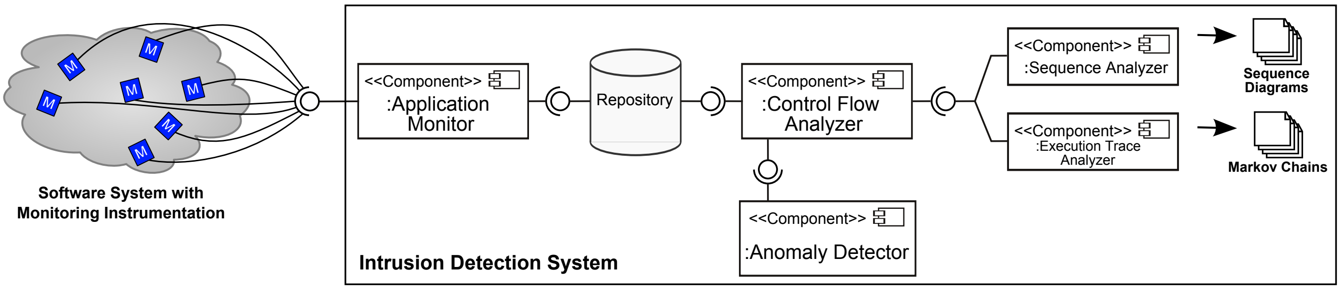
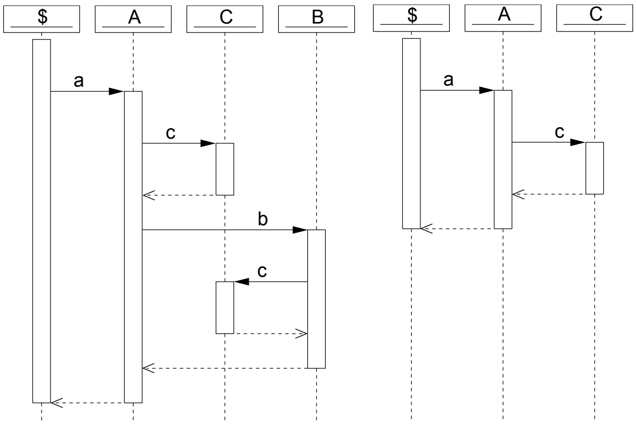


Fig. 1. Conceptual Architecture.



(a) Traces 1 and 2

(b) Trace 3

Fig. 4. Sequence diagrams of the example traces.

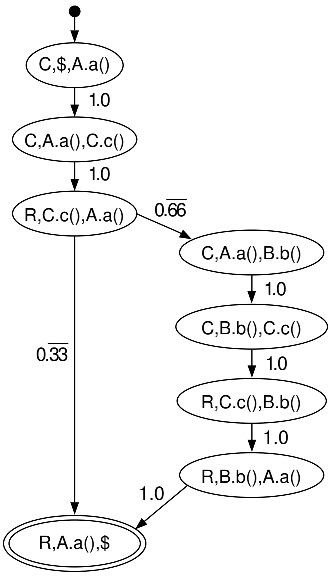


Fig. 5. Markov chain resulting from Traces 1, 2, and 3.

Next, the state machines are combined into a Markov chain. For this, a state machine union algorithm is used that iteratively merges states until some stop criteria is fulfilled. The resulting state machine is deterministic. There are various alternative algorithms for creating Markov chains from single state machines in the classical literature on grammar inference [11]. These differ in criteria such as whether the resulting underlying state machine only accepts the traces that are equivalent to the traces in the training data or whether generalization operations (e.g., related to loops) are performed. In our approach, we used a conservative strategy: the Markov chain only models those traces which are exactly equivalent to the traces captured in the learning phase. In order to prevent that this leads to a too restrictive intrusion detection, a sufficiently large amount of training data has to be provided, which contains each non-malicious trace at least once. The Markov chain's transition probabilities are computed according to the relative frequencies within the set of traces. A Markov chain for the three test cases is shown in Figure 5.

B. Creation of the Final Application Behavior Profile

The Markov chain created by the approach above can be considered a stochastic model for the normal application control flow for user requests. However, for systems in which much variance is possible within the traces, the resulting trace model can become very large. Therefore, a single huge Markov chain can lead to the acceptance of too many traces as normal behavior and to an unacceptable false negative rate (undetected intrusions). In the following, two clustering approaches are presented that both decrease the the size of the Markov chains and improve the detection quality. Both approaches are compared in the evaluation in Section VI.

The clustering works as follows. Agglomerative clustering [12] is used, i.e., the most similar trace clusters are iteratively merged until some similarity threshold is exceeded. The Levenshtein distance [13] (also known as the edit distance) is used to quantify the similarity between two traces. It is defined as the minimal number of edit operations (e.g., delete, add, replace element) that have to be performed to transform one sequence (a trace in our case) into another. In the case study, a second distance metric was used that clusters traces based on the corresponding request URL (not shown in the data schema here).

Equation 1 defines how the similarity between two traces t_1 and t_2 is computed based on the Levenshtein distance (Ld). The value $|t|$ denotes the length of a trace in terms of the number of messages.

$$similarity(t_1, t_2) := \frac{max(|t_1|, |t_2|) - Ld(t_1, t_2)}{max(|t_1|, |t_2|)} \quad (1)$$

For the running example, two additional traces (Figure 6) are added to demonstrate the clustering. First, the similarities between the traces are computed, as listed in Table II. Next, the agglomerative clustering uses the similarities to form clusters. For this example, the threshold λ is set to 0.65, which results in the three clusters illustrated in Figure 7.

Trace 4:
(C,$,A.a()) , (C,A.a(),C.c()) , (R,C.c(),A.a()),
(C,A.a(),D.d()) , (R,D.d(),A.a()) , (R,A.a(),$)

Trace 5:
(C,$,A.a()) , (R,A.a(),$)

Fig. 6. Two additional example message traces.

TABLE II
EXAMPLE: TRACE SIMILARITY (AND LEVENSHTEIN DISTANCE).

	1	**2**	**3**	**4**
5	0.25 (6)	0.25 (6)	0.5 (2)	0.$\overline{3}$ (4)
4	0.5 (4)	0.5 (4)	0.$\overline{6}$ (2)	
3	0.5 (4)	0.5 (4)		
2	1 (0)			

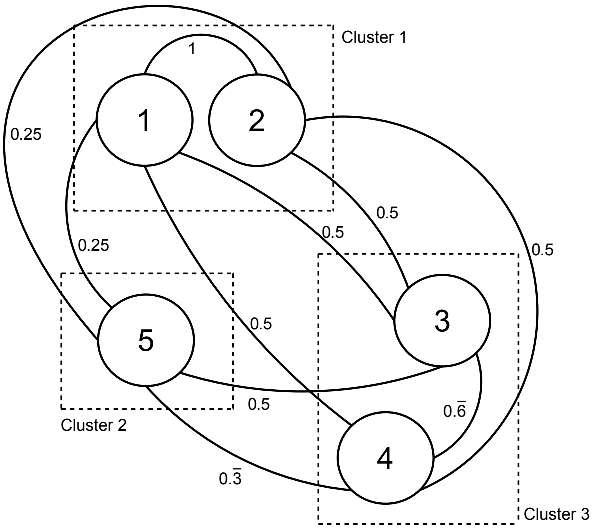


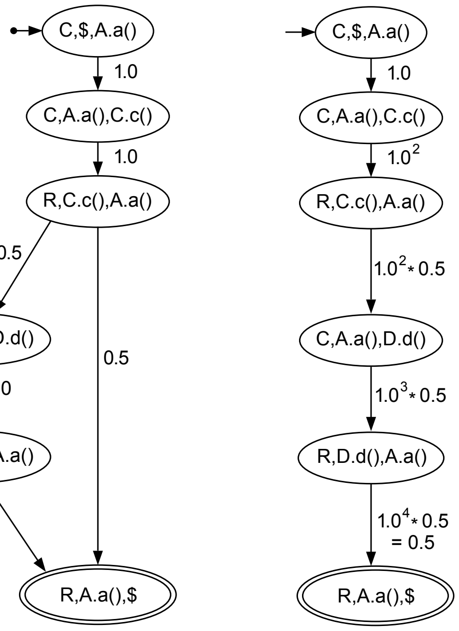
Fig. 7. Three clusters are generated from the five example traces.

V. ANOMALY DETECTION

Anomaly detection is triggered, when newly completed traces are discovered in the repository. The process of anomaly detection can be described as follows.

Let t be a trace that corresponds to a new request received by the Web application. Given the normal application behavior profiles as the set of n Markov chains $MC = \{m_1, m_2, ..., m_n\}$, an anomaly is detected if there is no $m \in MC$ which accepts t with a probability higher than a previously defined threshold τ. Formally, the maximal product of the probabilities along the pathes corresponding to trace t through each Markov chain has to be below τ to flag t an intrusion. In other words, t is flagged an anomaly if the best fitting Markov chain to t shows a probability below τ for trace t.

Afterwards, the anomaly detector checks the incoming traces against all Markov chains until a Markov chain with a corresponding path of sufficient probability τ matching the trace is found. If no such path can be found in all Markov chains, a possible intrusion attempt is signaled. One Markov chain for each cluster is generated. The Markov chain for cluster 3 is shown in Figure 8.



(a) Markov chain of cluster 3.

(b) Computation of the anomaly score for trace 4

Fig. 8. Markov chain of cluster 3 (8(a)) and computation of trace 4's anomaly score (8(b)). For a threshold $\tau > 0.5$, trace 4 is considered an intrusion.

VI. EVALUATION

An intrusion detection system has two important quality attributes: the rate of false positives (false alarms) and the rate of false negatives. The goal of this case study is to test the general applicability of the approach and to study these both quality attributes for several intrusion scenarios in the context of a sample Web application.

The software system under supervision in the case study is the iBATIS JPetStore[2], which is a demo Java Web application implementing an online store scenario. All 199 internal operations of the JPetStore were instrumented. In the following,

[2] http://ibatis.apache.org/

first the creation of the normal behavior profile and the trace clustering are described before the intrusion scenarios are presented and used to evaluate the intrusion detection approach.

A. Training Data for the Normal Application Behavior Profile

The training data consists of more than 1,000,000 executions from 14,194 monitored traces. Most of these traces have been generated automatically by using the probabilistic workload generator Markov4JMeter[3] [14], which is an extension of Apache JMeter. Several hundred additional traces were generated by manual application usage.

B. Construction of the Normal Application Behavior Profile

Four different clustering variants have been used for creating the normal behavior profile. Three of them use the similarity metric given in Equation 1 that uses the Levenshtein distance with different clustering thresholds. The fourth clustering variant forms clusters based on the URL that corresponds to the trace's request. A brief description of the clusters is as follows:

$\mathbf{C}_{000}$: No clustering (equivalent to clustering with threshold $\lambda = 0.0$) results in a single large Markov chain.

$\mathbf{C}_{060}$: Clustering with threshold $\lambda = 0.60$ results in 30 Markov chains.

$\mathbf{C}_{080}$: Clustering with threshold $\lambda = 0.80$ results in 54 Markov chains.

$\mathbf{C}_{\mathrm{URL}}$: Clustering based on the request URL results in 25 Markov chains.

C. Intrusion Scenarios

Intrusion scenarios were designed to find vulnerabilities, e.g., possible SQL injections in custom Web applications as attackers would be interested in the information stored in the database, e.g., credit card numbers and personal user information. The procedure of an attacker would be to check the parts of an application which takes user input for SQL injection vulnerabilities. An intrusion was signaled each time an SQL injection attempt was made.

We conducted a security audit of the iBATIS JPetStore to discover intrusion scenarios. This resulted in five potential vulnerabilities that could be used in an attack. The first three of the vulnerabilities are due to the possibility to raise unhandled exceptions, and the fourth is the possibility to order a negative quantity of products, which also affects the total cost calculation for an order. The last intrusion attempts a direct access to user data in a way that tries to bypass a proper login.

The JPetStore uses an SQL database to store business data and critical information such as credit card numbers. Hence, SQL injection attacks might be possible. However, no direct SQL injection vulnerabilities were discovered but SQL injection attempts resulted in the three unhandled exception scenarios.

D. Intrusion Detection Results

Table III lists the intrusion detection quality results for the experiments. The SQL injection attacks had enough effects on the application control flow to allow our system to detect the intrusion attempts.

The profiles C_{000} and C_{060} produced no false positives at all, while C_{080} and C_{URL} produced false alarms in 5.5% of the runs. These 5.5% correspond to runs in which a user tried to remove an item from an empty cart. This test case was not covered by the automatic workload generation. Similar cases were generated during the manual browsing of the system for normal behavior creation. The 20% for C_{060}, C_{080}, and C_{URL} false negatives occur because the intrusion scenario that represents an order of a negative quantity of products does not effect the application control, and hence it is not detectable by control flow analysis. The single Markov chain for C_{000}, fails both to detect the negative quantity order and to detect the last intrusion scenario (direct data access), since it tends to overgeneralize allowed behavior from the training data. The anomaly detection threshold τ fixed to 0.025 in all experiments.

TABLE III
INTRUSION DETECTION RESULTS.

	False pos. rate	**False neg. rate**	**Cluster count**
0.00	0%	40%	1
0.60	0%	20%	30
0.80	5.5%	20%	54
URL	5.5%	20%	25

VII. Related Work

Denning [3] describes a model capable of detecting break-ins and penetrations by monitoring system audit records for abnormal behavior. The author suggests to monitor the system using audit records which represent actions performed on objects by subjects. Anomalies in this audit records indicate a security breach.

Hofmeyr et al. [7] followed a similar approach by detecting anomalies in UNIX system call traces for intrusion detection. The classification of anomaly is made by means of string comparisons of system call sequences.

Lane and Brodley [15] focus on four challenges of intrusion detection: the classification of normal behavior is often user- and site-dependent; the learning of the normal behavior must happen on positive data only; the selection of the characteristics that are used to build a model of normal behavior and incorporation of changing user behavior.

ModSecurity[4] is a freely available host-based intrusion detection system for Web applications running in the Apache Web server. It combines a pattern-based with an anomaly detection based approach in order to cover a large variety of attacks. It allows the administrator to define white list patterns

[3] http://markov4jmeter.sourceforge.net

[4] http://www.modsecurity.org/

Appeared in Proceedings of the 12th IEEE International Multitopic Conference (IEEE INMIC 2008), pages 373–378. IEEE. Dec. 2008.

and black list patterns for user input. White list patterns are the preferred way of securing an application, because the evasion of white list filter rules is not possible. Whereas with black list filters it is always possible that the administrator misses some corner cases or that new ways of attacks against systems emerge [16].

Chari and Cheng [17] introduce the host-based intrusion detection system BlueBoX. This system monitors the application behavior and checks it against predefined rules. The rules specify the actions which an application is allowed to do. Attempts to violate this rules are denied and flagged as an intrusion attempt. The effectiveness of this policy-based approach depends on the definition of high quality rules by the administrator. Though, it is hard to define good rules and the task of defining this rules needs a precise understanding of the expected system behavior.

Snort[5] is a pure pattern-based approach to network-based intrusion detection. It is described as a lightweight intrusion detection, whereby lightweight means that this system can be used cross-platform, with a small system footprint and that it is easily configurable by the administrator. Since Snort is a pattern-based intrusion detection system, it success depends on the quality of the supplied rules. The rules have to be updated regularly to detect new emerging attacks [18]. Snort is widely used in productions environments. Anomaly detection can be integrated into Snort by using the SPADE [6] plug-in.

VIII. Conclusions

In this paper, we presented a novel intrusion detection approach based on application-internal control flow analysis. It was demonstrated that it can potentially detect attacks against typical vulnerabilities, and therefore could provide a valuable addition to the security of software systems. An additional clustering step prevented that only a single, too general normal behavior model was produced.

Future work includes to combine application behavior monitoring with user behavior monitoring to create a normal behavior profile. Additionally, instead of using only control flow monitoring for every user request, the approach will be extended to connect the sequence of all operation executions within a complete user session. A case study in an industry system is currently in preparation in order to determine the intrusion detection quality in the field.

References

[1] "12th annual computer crime and security survey," Computer Science Institute, Federal Bureau of Investigation, 2007.

[2] S. Kumar, "Classification and detection of computer intrusions," Ph.D. dissertation, Purdue University, 1995.

[3] D. Denning, "An intrusion-detection model," *Transactions on Software Engineering*, vol. 13, no. 2, pp. 222–232, Feb. 1987.

[4] D. C. Schmidt, "Middleware for real-time and embedded systems," *Commun. ACM*, vol. 45, no. 6, pp. 43–48, 2002.

[5] P. Ning and S. Jojodia, "Intrusion detection techniques," in *The Internet Encyclopedia*. John Wiley & Sons, 2003, pp. 355–367.

[6] S. Staniford, J. A. Hoagland, and J. M. McAlerney, "Practical automated detection of stealthy portscans," *J. of Comp. Sec.*, vol. 10, pp. 105–136, 2002.

[7] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, Aug. 1998.

[8] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for UNIX processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy (SP '96)*. IEEE, 1996, p. 120.

[9] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring, "Kieker: Continuous monitoring and on demand visualization of Java software behavior," in *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE 2008)*. ACTA Press, Feb. 2008, pp. 80–85.

[10] Object Management Group (OMG), "Unified Modeling Language: Superstructure Version 2.1.1," Feb. 2007.

[11] R. Solomonoff, "A formal theory of inductive inference, I and II," *Information and Control*, vol. 7, pp. 1–22 and 224–254, 1964.

[12] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 264–323, 1999.

[13] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, Feb. 1966.

[14] A. van Hoorn, M. Rohr, and W. Hasselbring, "Generating probabilistic and intensity-varying workload for Web-based software systems," in *Proceedings of the SPEC International Performance Evaluation Workshop 2008 (SIPEW '08)*, ser. LNCS, vol. 5119. Springer, Jun. 2008, pp. 124–143.

[15] T. Lane and C. E. Brodley, "Detecting the abnormal: Machine learning in computer security," Department of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907, Tech. Rep., Jan. 1997.

[16] Breach Security, "ModSecurity reference manual (version 2.1.0)," 2007. [Online]. Available: http://www.modsecurity.org/documentation/modsecurity-apache/2.1.3/modsecurity2-apache-reference.pdf

[17] S. N. Chari and P.-C. Cheng, "Bluebox: A policy-driven, host-based intrusion detection system," *ACM Transactions on Information System Security*, vol. 6, no. 2, pp. 173–200, 2003.

[18] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX Conference on System Administration (LISA'99)*. Berkeley, CA, USA: USENIX Association, 1999, pp. 229–238.

[5] http://www.snort.org/