A Dataset Generator for Next Generation System Call Host Intrusion Detection Systems

Marcus Pendleton and Shouhuai Xu
Department of Computer Science, The University of Texas at San Antonio, San Antonio, TX 78249 USA

***Abstract*—Over the years, system calls (syscalls) have become an increasingly popular data source for host intrusion detection systems (HIDS). This is partly due to their strong security semantic implications. As syscalls conform to a program's control-flow graph, a deviation in a syscall sequence may imply a deviation in a program's control-flow graph. This is useful for detecting the control-flow hijacking class of attacks. Additionally, malware must utilize syscalls in order to provide any utility to the attacker, with the exception of some denial-of-service attacks. Because all syscalls are observable from the kernel, this makes evasion difficult for attackers under syscall HIDS. Given their suitability for HIDS, many approaches based on syscalls have been proposed. However, the syscall datasets available are not always the most suitable for these and emerging techniques in analytics, as they may need additional structural or contextual information about syscalls in their decision engine. Furthermore, this flatness of previous datasets often pigeonholes solutions into those which are limited by that data view. It is also burdensome on the researcher to generate his own custom dataset. In this work, we propose an extensible syscall dataset generator which includes structural and limited contextual information regarding syscalls, yet allows for researchers to easily add their own features to more quickly develop and evaluate their systems. Our dataset generator can aid researchers in widening the solution space for syscall HIDS.**

***Index Terms*—Dataset, Intrusion Detection, Linux, Operating Systems, Security, System Calls.**

I. Introduction

Important to research and development of system call (syscall) HIDS are syscall datasets with which a proposed system can be validated and compared with others. Prior availability of syscall datasets expedites development of syscall HIDS, as time can be focused on the decision engine (DE) rather than the development of yet another syscall dataset. However, current datasets for syscall HIDS contain only flat sequences or lack contextual information, the problems of which will be elaborated in Section III. This pigeonholes solutions into those which deal only with such sequences, limiting the success that can be achieved with syscall HIDS. Additionally, these datasets are derived from very simple programs, many of which are single-threaded [9]. Therefore, results against these datasets can be misleading, as the syscall sequences contained within are not representative of the many complex programs that are highly vulnerable to attacks and of high interest to defenders (e.g., web browsers and web servers).

In this work, we introduce a dataset generator that 1) provides structural and contextual information from an arbitrary target program, 2) is extensible to include additional execution-feature a researcher deems important for anomaly detection, and 3) is public domain to encourage future development. The overarching goal is to widen the solution space of syscall HIDS and expedite their development. The rest of this paper is organized as follows: Section II briefly describes the related work, Section III underpins the fundamental limitations of previous datasets, Section IV describes the basic ideas the derived objectives for our generator, Section V discusses implementation details, and Section VI concludes.

II. Related Work

Three main datasets exist with which researchers have been testing their syscall HIDS: The Knowledge Discovery and Data Mining '98 and '99 datasets (KDD), The University of New Mexico dataset (UNM), and the more recent Australian Defence Force Academy Linux Dataset (ADFA-LD) [1] [2] [4] [9]. In the following subsections, a characterization of each dataset is given, highlighting their pros and cons.

A. The KDD Datasets

The KDD '98 and '99 datasets were used for The International Knowledge Discovery and Data Mining Tools Competition, held in conjunction with The International Conference on Knowledge Discovery and Data Mining. They are composed of Solaris BSM audit logs, which contain a plethora of system-wide events in addition to syscalls for analysis. These datasets represent the first systematic, valuable and innovative resources for the early development of syscall HIDS. Forrest et al introduced her seminal work in syscall anomaly detection in [12] with evaluations against these datasets. Her contributions continues to influence the use of syscalls as a means of anomaly detection today.

Despite being among the first datasets with which syscall HIDS were developed, they have been heavily criticized for their use in evaluating more recent syscall HIDS. [16] [17] [11] and [16] highlight that the 90s era systems under which the data was collected hardly reflect the systems in use today and critique the methodology used to generate the datasets. Finally, little emphasis is placed on the structure of syscall sequences and absolutely no CPU-context information about the syscalls are contained within the logs. The problems associated with these deficiencies will be elaborated in Section III.

978-1-5386-0595-0/17/$31.00 ©2017 IEEE

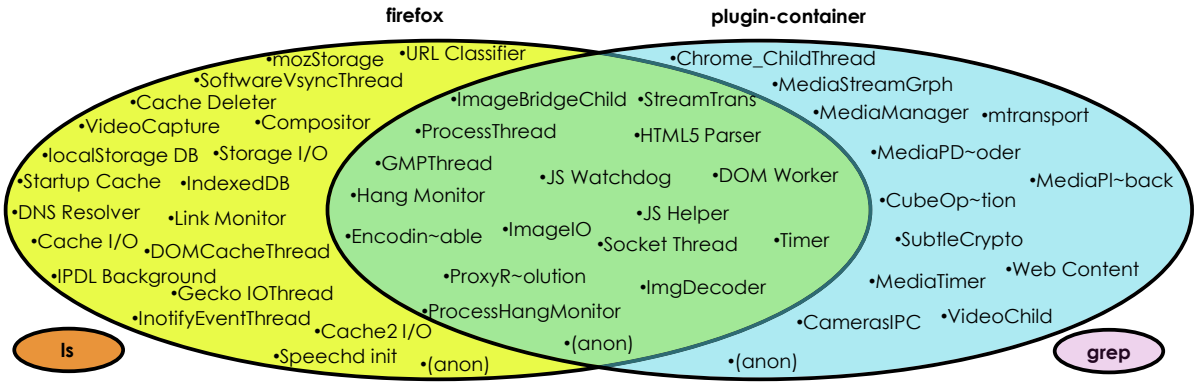


Fig. 1: Identified Thread Functions in Mozilla Firefox

B. The UNM Dataset

The UNM dataset was popularized by the groundbreaking work of Forrest et al [12] [20]. It contains syscall sequences generated in SunOS from the programs `lpr`, `xlock`, `named`, `login`, `ps`, `inetd`, `sendmail`, and `s-tide`, which is a syscall HIDS itself. The sequences are from both synthetic and live sources.

The variety of programs from both synthetic and live sources made this a good dataset for early work in syscall HIDS. However, the programs are simple in comparison to highly complex and dynamic programs (e.g., with dynamically loadable modules) such as web servers and browsers. Good results against this dataset could be misleading, as they may fail dramatically against more complex and realistic datasets [9]. Additionally, more elaborate DEs need more sequences to converge, and the relatively few sequences split among the various programs in the UNM datasets may be largely insufficient for more capable machine learning algorithms. Finally, the syscall sequences are also flat.

C. The ADFA-LD Dataset

The ADFA-LD dataset was designed to succeed the previous two datasets, with an emphasis on machine-wide collection of syscalls. Data was collected using the `auditd` program under Linux. Aside from the data generation on a more modern system, it contains a rich set of attacks ranging from password cracking to web shells.

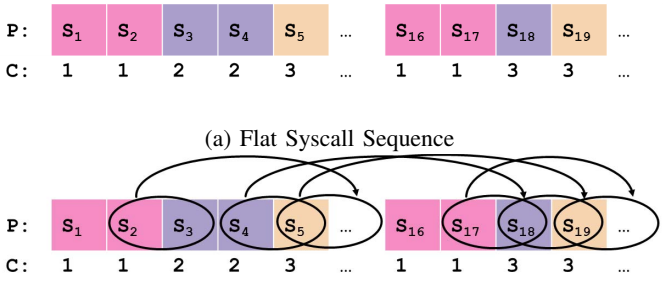
Unfortunately, ADFA-LD suffers from the flatness as previous datasets. With sequences such as $6, 6, 63, 6, 42, \ldots$,no structure or context can be obtained from the syscalls. The implications of this will be explained in the next section.

III. Problem

A quick glance at previous datasets reveals a few problems. These can be characterized by the structure, type and quantity of the syscall sequences and the complexity of the target programs they profile. The following sections elaborate on these problems and highlight their implications on intrusion detection.

A. Structure

The most limiting characteristic of previous datasets is the structure of their syscall sequences. All sequences in these datasets are flat: linear sequences with little to no thread information. In the case of multi-threaded programs, which arguably comprise the bulk of programs defenders want to monitor, this poses a problem. In particular, the syscalls generated from the parent and various children threads of a program are interleaved in a linear sequence without distinction. Figure 2a depicts a flat sequence of syscalls. As syscall sequence recognition is essentially a language modeling problem [18], this degrades the learned model as "bad transitions" (circled in Figure 2b) increase the set of acceptable languages considered normal. This can be exploited by attackers as this gives him more maneuverability in the set of malicious sequences he can generate that may evade a DE in a mimicry attack [19], illustrated in Figure 6. Shaded areas represent area for maneuverability in a mimicry attack.



(a) Flat Syscall Sequence

(b) Bad Transitions (circles) and Dependence (arcs)

Fig. 2: Flat Syscall Stream and Associated Problems

1) Non-Determinism: In the event that two or more threads execute syscalls simultaneously (e.g., in a multi-core system), their order in a flat sequence is uncertain. Furthermore, as thread execution is non-deterministic with respect to identical program inputs, and depends on system load and scheduling algorithms, so are the resulting flat syscall sequences. Figure 3 depicts this phenomenon, where a program P is executed

multiple times with the same input. This demands that HIDS algorithms that use flat inputs accommodate this phenomenon, with adjacent syscall events possibly resulting from different threads in a non-deterministic fashion. These "bad transitions" represent deviations in a control-flow graph (CFG), thus allowing derived syscall HIDS models to permit them. This is one source of inaccuracy in existing syscall HIDS. In Figure 2b, this phenomenon is captured by circled adjacent syscalls, where each syscall in a pair is generated by a different thread. Popular subsequence database (DB) and hidden Markov model (HMM) methods directly learn these bad transitions.

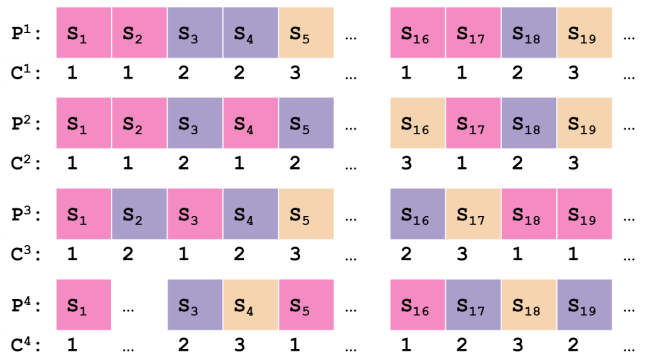


Fig. 3: Non-Determinism in Syscall Sequences

2) Dependence: Related to the problem of interleaved syscalls is the challenge of learning dependence. In this context, a dependence refers to the true, per-thread order of syscalls. However, in flat sequences, this true order is interjected with syscalls from all threads (parent and children). The arcs in Figure 2b depict true dependence to highlight the correct transitions a model must learn, and interjected threads disrupt this order. A HIDS algorithm must learn to discover these dependences in a flat sequence to more accurately model the set of acceptable syscall streams a program may generate.

Subsequence DBs and HMMs (for the most part), cannot capture these dependences. In special cases of HMMs, higher-order models predicting states from the previous n states can be modeled, but incurs an exponential growth in states with respect to n. Currently, the best class of models for learning dependence using flat streams is recurrent neural networks, with Long Short-Term Memory (LSTM) networks performing the best among them [14]. However, due to non-determinism, these networks require many presentations of equivalent sequences to discover dependences.

B. Type

Another restricting characteristic of previous datasets is that the syscall sequences are purely numbers indicating which kernel service is desired by the user space program. In other words, the context of syscalls is omitted. Research such as [15] shows that syscall arguments can also be used for intrusion detection. A dataset that includes such context information (e.g., register values and limited memory operands) in addition to pure syscall numbers may lead to the development of more accurate syscall HIDS.

C. Quantity

Previous datasets give a limited number of training sequences to build DEs. This is a strong a assumption on the adequate number and length of sequences needed for new techniques. As machine-learning approaches grow in complexity, so does the quantity of training data they require. This is especially true in the case of LSTM neural networks [14]. Typically a large number of machine-learning parameters is necessary to model a language, especially a potentially complex syscall language of a program. Figure 5 shows how a CFG can be converted into a syscall finite-state machine (FSM) using modified Depth-First Search (DFS) [13], [18], [21]. Consequently, the complexity of the FSM, and it corresponding regular language, is dependent on the CFG. The number of parameters determines the amount of training data to derive a sufficient model. As a result, more training data than what is currently offered by current datasets may be insufficient for new approaches.

D. Complexity of Targets

The aforementioned problems become exacerbated as the complexity of a program increases. In this work, complexity refers to the number of distinct syscalls and thread functionalities in a program. Thread functionalities correspond to distinct subgraphs in a whole-program CFG (super CFG) which individual threads traverse. Previous datasets, with the exception of ADFA-LD, profile simple, less complex programs such as `lpr`. Syscall HIDS which report success with such programs are misleading as they fail with complex, real-world programs of interest to defenders due partly to the previously discussed problems [9] [8] [10]. ADFA-LD falls short in that, although it contains sequences generated by complex programs, its sequences are flat and contain no thread and context information. Finally, none of the previous datasets address the fact that complex programs, or attacks, may spawn processes which generate sequences outside of the target. It is necessary to incorporate the sequences of spawned processes to aid in verifying 'helper' programs or detecting malicious activity such as unauthorized shell execution. Mozilla Firefox is an example of an ideal target for syscall HIDS validation as it is highly complex with helper programs, plugins (dynamically loadable code) and numerous distinct syscalls [3]. Figure 1 illustrates the high degree of thread functionality in Mozilla Firefox, with each set corresponding to a 'helper' program, and members in each set referring to child thread functionalities. These thread functionalities were identified using the `comm` attribute in the Linux data structure `struct task`, which developers of Mozilla Firefox presumably set for debugging purposes, as this field is usually the name of the executable. During the course of normal execution, the main binary of Mozilla Firefox (`firefox`) may spawn `plugin-container`, `ls` and `grep`. `firefox` and `plugin-container` can execute up to 40 and 29 identifiable thread functionalities (excluding anonymous threads), respectively, each of which generates and interjects syscalls into a flat syscall stream non-deterministically. The overlap of

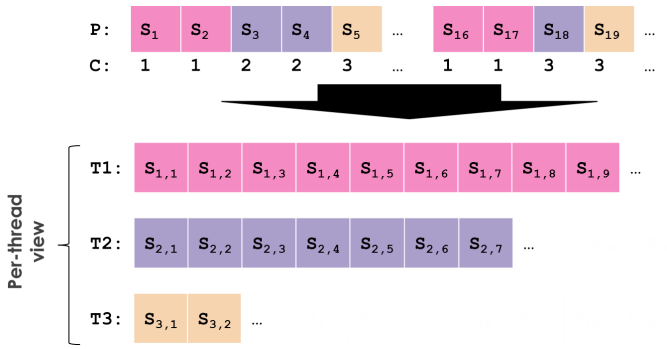
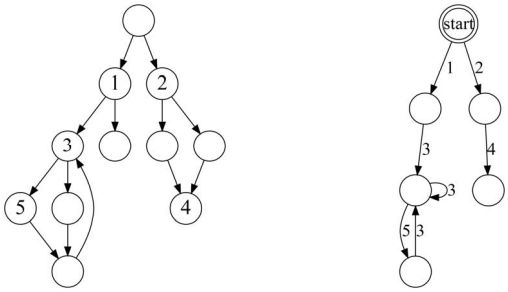


Fig. 4: Flat to Per-Thread Sequences for Modeling



(a) Annotated CFG (b) Derived Syscall FSM

Fig. 5: CFG to FSM (Regular Language) Conversion via Modified DFS. Numbered Basic Blocks Contain Syscalls

`firefox` and `plugin-container` in this figure refers to functionality that may be common to both, perhaps provided by shared libraries.

IV. Basic Ideas

As mentioned earlier, the syscall HIDS can be reduced to a language modeling problem. Therefore, a dataset generator focused on providing structured (filtered) per-thread streams of syscalls to avoid the problems of flat, interleaved, noisy sequences will help syscall HIDS learn languages more representative of their respective programs. As such, this improved view on syscall streams is the basis of this work. The transformation of a flat sequence to a structured sequence is illustrated in Figure 4. In the following subsections, we elaborate on the implications of flat and structured syscall streams on language modeling, and outline derived objectives for a better dataset generator to aid in improved syscall HIDS design and development.

A. Languages of Syscall Sequences

The problem of syscall HIDS can be viewed as a language modeling problem [19]. That is, given a dataset S of program sequences, or the control-flow specification M extracted from source or binary (Figure 5), the derived language accepts a set of sequences representative of normal program behavior. Given a dataset S (or model M), a language $L(S)$ ($L(M)$) is the collection of sequences that a DE will recognize as acceptable. With the aforementioned problems of using flat sequences in model building, the language L can accept considerably more sequences not representative of normal program behavior. Refer to Figure 6 for an illustration of this. $L(True)$ represents the ideal language that a DE should detect; syscall sequences only representative of those a multi-threaded program can generate conforming to its CFG. $L(Flat)$ represents the language, based on the same program, in which syscalls from multiple threads are interleaved. In this example, the model accepts more sequences than it should, providing an attacker more freedom to construct malicious sequences deemed as acceptable by a DE. $L(PerThread)$ represents a language closer the $L(True)$ which we expect to achieve by using per-thread streams for model building and analysis, as will be elaborated below. $L(PerThread)$ will also have shortcomings attributed to the selected HIDS algorithm, such as HMM's probabilistic nature, thus leaving room for future improvement by utilizing other features such as syscall arguments, limited memory operands, etc. This language is denoted in the figure by $L(PerThread + Context)$.

TABLE I: Comparison of Dataset/Generator Features

	UNM	KDD	ADFA-LD	Ours
Synthetic Source	yes	no	no	yes
Live Source	yes	no	no	yes
Thread Info	no	no	no	yes
Context Info	no	no	no	yes
Timing Info	no	no	no	yes
Spawn Following	no	no	no	yes
Extensible	no	no	no	yes
# of Programs	9	980	-	-

B. Dataset Objectives

Given the problems of previous datasets, we set out to design a generator to lift the restrictions they impose on conceiving, developing and validating methods leveraging new techniques in analytics. The main objectives of the generator are listed below:

- Thread-Sensitivity: to allow the isolation of syscall patterns per thread, resulting in more tailored models
- Context-Sensitivity: to disambiguate the multiple roles a syscall may serve in a program (e.g., `write` for file or socket)
- Unlimited Sequences: to remove assumptions on the number of sequences needed to train, validate and test a system.
- Complex Targets: to allow for more realistic evaluation of a HIDS
- Extensibility: to allow researchers to provision additional execution features to their HIDS.
- Public Domain: to encourage future expansion and development of the dataset and collection system (git@github.com:marcusp46/syscall-dataset-generator.git)

Table I shows a comparison between qualities of previous datasets versus those of datasets output from our generator. In Section V, implementation details for the generic compo-

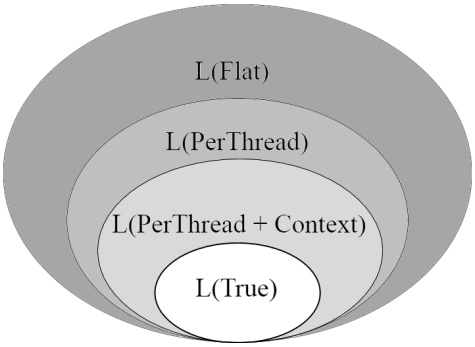


Fig. 6: Various Syscall Languages of a Program: Flat, PerThread, PerThread+Context, and True

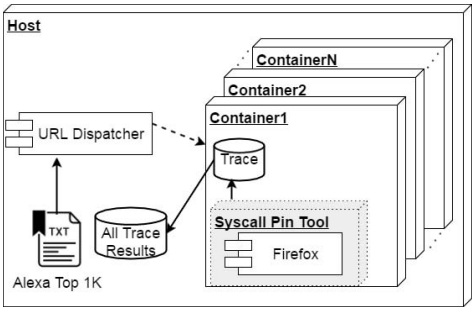


Fig. 7: Architecture of Client-Based Target Generator

nents and the configuration for profiling Mozilla Firefox are discussed.

V. Generator Architecture

In this section, we document the implementation details for the generator. It is important to note that the design is generic and intended to be adaptable to any target. As such, researchers can profile a number of programs to test their approaches. Mozilla Firefox was chosen as the target in this work mainly because of its very high complexity and named threads, which can be used to validate anonymous thread identification techniques. Techniques which are able to discriminate between normal and anomalous sequences generated from such a program may prove to be more viable for real-world deployment. Additionally, Mozilla Firefox is more representative of programs which defenders want to monitor, as browsers are frequently targeted as infection vectors for malware. Therefore, synthetic and attack sequences from real exploits can be generated. Figure 7 depicts an architectural overview of the Mozilla Firefox syscall generator, comprised of the syscall collector, a containerized target execution environment, and dispatcher.

A. Syscall Collector

The syscall collector is the most important component of the syscall dataset generator. It is responsible for collecting per-thread syscall sequences along with contextual information for each syscall executed. Additionally, it must be extensible per the objectives established earlier in Section IV. Intel's Pin was used as the collection mechanism by which these qualities are achieved [5]. Pin is a dynamic instrumentation tool which rewrites application code on-the-fly to include user-defined profiling and analysis routines and provides a rich API for accessing rich execution information. Pin is also available for Linux and Windows platforms, meaning that the generator can be used to develop syscall HIDS for both systems. For demonstration purposes, Ubuntu Linux is the host platform for profiling Mozilla Firefox [7].

a) Spawn (execv) Following: As mentioned earlier, not only is tracing the syscalls executed within a particular target important, but also those of programs spawned by that target. This allows for a more comprehensive analysis of a target's activity, as the target may use the services of other programs. This capability of processes can just as easily be exploited to invoke programs of interest to attackers. In the case of Mozilla Firefox, additional instances of `firefox`, as well as instances of `plugin-container`, `grep` and `ls` may be spawned during the course of normal execution (Figure 1). As such, the syscall collector follows spawns by tracking the `execv` class of syscalls, which are responsible for programs executing other programs. For each spawned program, syscalls are tracked in the same manner as those from the parent process. Details of the syscall collector's output are explained in Section V-A0b.

b) Output: The output of the syscall collector can include any user-defined execution feature appropriate for a particular DE. However, the default output of our system provides baseline output to yield the desired qualities of syscall sequences previously discussed as well as flat sequences for DE comparison purposes and a reduced fileset size. Listing 1 describes fileset generated from an execution trial of a target. *execve* refers to the base filename of the main and spawned executable files, *iteration* refers to the invocation instance of those executable files (zero-based), and *ID* refers to the internal identification of threads according to order of creation within a process (zero-based).

1) `flat _<execv_name>_<iteration>.out`
   - flat program-view sequence to serve as input into legacy approaches and correlation to context and scheduling info for per-thread sequence reconstruction
2) `context_<execv_name>_<iteration>.out`
   - flat context info (register arguments) of each syscall corresponding to positions in 1
3) `schedule_<execv_name>_<iteration>.out`
   - identifier of thread contexts corresponding to positions in 1
4) `timing_<execv_name>_<iteration>.out`
   - inter-arrival time of syscalls corresponding to positions in 1
5) `thread_names_<execv_name>_<iteration>.out`
   - line-by-line listing of thread names indexed by *ID* string)
6) `temporal_<ID>_graph_<execv_name>_<iteration>.out`
   - per-thread temporal graph data structure of syscall execution
7) `thread_<ID>_<execv_name>_<iteration>.out`
   - per-thread streams of syscall sequences
8) `summary.out`
   - statistics of syscall execution during trial

Lst. 1: Output Files of The Generator and Their Descriptions

It is important to note that the flat sequence is meant not only for input into traditional syscall HIDS approaches,

but reduce the number of per-thread files output after an execution trial. For example, per-thread inter-arrival times can be reconstructed using the `flat*`, `scheduling*` and `timing*` files.

B. Containerized Execution Environment

The containerized execution environment leverages features of Docker, which offers some of the isolation benefits of virtual machines (VM) in terms of segregating processes, filesystems and network interfaces [6]. In contrast to VMs, containers share the same kernel as the host but have a significantly lower resource footprint (CPU, memory) and much shorter provisioning time. Similar to VMs, containers are provisioned from images, which are typically mutable and changes are kept local to an instance, discarded upon container termination. This allows for the target program to run in a fresh environment in many instances simultaneously on a host, increasing the throughput of execution trials for dataset generation. Additionally, containers can be programmatically managed for full automation.

C. Dispatcher

The dispatcher coordinates all activities of the generator. For profiling Mozilla Firefox, it first receives input for URLs to visit. The source of URLs is discussed in Section V-D. For each URL, the dispatcher provisions a container to run Mozilla Firefox under the syscall collector, storing the output files locally on the container. At termination, the files are copied from the container to the host under a unique identifier for the visit event.

D. Benign and Attack Datasets

For syscall HIDS development, benign and attack syscall sequences are necessary to train, validate and evaluate the derived models. One approach is to use controlled environments with automated interactions with the target program to generate synthetic data. The benefit of this approach is that known-benign and known-malicious interactions can be crafted. The limitation in this approach is that it is burdensome to create interactions rich enough to discover as much thread functionality as possible to reduce false-positives when tested in real-world environments. In the context of Mozilla Firefox, it incumbent on the developer to create test websites with diverse content to explore as much of the code regions as possible. The other approach is to direct the target to interact in live environments. Although it is easier for the developer to discover more thread functionality, ensuring whether a source is benign or malicious is a challenge. The generator can be used for both, however, we followed the lead in [22], where Xu et al used the top 1K popular websites for benign sources. Attack sequences were generated in a controlled environment.

VI. Conclusion

Syscalls have yet to realize their full potential in anomaly detection. Research has focused on datasets which lack much of the information about syscall events that can be utilized to improve detection accuracy. Also, the development of new datasets can be very time consuming.

This paper presents a dataset generator that gives researchers and syscall HIDS an improved view of syscall sequences. It will guide researchers in discovering novel ways to utilize syscall sequences for HIDS and expedite their development. Our solution's rich dataset generation, extensibility and availability to the public under license should lead to more exchange of ideas.

Acknowledgement. This research was supported in part by ARO Grant #W911NF-13-1-0141 and NSF Grant #1111925.

References

[1] "Kdd98 intrusion detection dataset," Online, 1998. [Online]. Available: http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/
[2] "Kdd99 intrusion detection dataset," Online, 1998. [Online]. Available: http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/
[3] "Mozilla firefox," https://www.mozilla.org/firefox/, 2002–2017.
[4] "University of new mexico intrusion detection dataset," 2004. [Online]. Available: http://www.cs.unm.edu/ immsec/systemcalls.htm
[5] "Intel pin," https://software.intel.com/sites/landingpage/pintool/, 2005–2017.
[6] "Docker," https://www.docker.com/, 2013–2017.
[7] "Ubuntu linux," https://www.ubuntu.com/, 2014–2017.
[8] G. Creech, "Developing a high-accuracy cross platform host-based intrusion detection system capable of reliably detecting zero-day attacks," Ph.D. dissertation, PhD thesis, University of New South Wales, 2014.
[9] G. Creech and J. Hu, "Generation of a new ids test dataset: Time to retire the kdd collection," in *2013 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2013, pp. 4487–4492.
[10] ——, "A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns," *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 807–819, 2014.
[11] V. Engen, J. Vincent, and K. Phalp, "Exploring discrepancies in findings obtained with the kdd cup'99 data set," *Intelligent Data Analysis*, vol. 15, no. 2, pp. 251–276, 2011.
[12] S. Forrest, "A sense of self for unix processes," *Security and Privacy*, 1996.
[13] J. T. Giffin, S. Jha, and B. P. Miller, "Detecting manipulated remote call streams." in *USENIX Security Symposium*, 2002, pp. 61–79.
[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.
[15] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," in *European Symposium on Research in Computer Security*. Springer, 2003, pp. 326–343.
[16] M. V. Mahoney and P. K. Chan, "An analysis of the 1999 darpa/lincoln laboratory evaluation data for network anomaly detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2003, pp. 220–237.
[17] J. McHugh, "Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 4, pp. 262–294, 2000.
[18] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 2001, pp. 156–168.
[19] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM, 2002, pp. 255–264.
[20] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*. IEEE, 1999, pp. 133–145.
[21] K. Wee and B. Moon, "Automatic generation of finite state automata for detecting intrusions using system call sequences," in *Computer Network Security*. Springer, 2003, pp. 206–216.
[22] L. Xu, Z. Zhan, S. Xu, and K. Ye, "Cross-layer detection of malicious websites," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 141–152.