2020 Second IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA) | 978-1-7281-8543-9/20/$31.00 ©2020 IEEE | DOI: 10.1109/TPS-ISA50397.2020.00037

A Modern Implementation of System Call Sequence Based Host-based Intrusion Detection Systems

Jeffrey Byrnes*, Thomas Hoang†, Nihal Nitin Mehta‡ and Yuan Cheng§
Department of Computer Science
California State University, Sacramento
Sacramento, CA, U.S.A.
*jeffreybyrnes@csus.edu, †thomashoang@csus.edu, ‡nihalnmehta@csus.edu, §yuan.cheng@csus.edu

***Abstract*—Much research is concentrated on improving models for host-based intrusion detection systems (HIDS). Typically, such research aims at improving a model's results (e.g., reducing the false positive rate) in the familiar static training/testing environment using the standard data sources. Matching advancements in the machine learning community, researchers in the syscall HIDS domain have developed many complex and powerful syscall-based models to serve as anomaly detectors. These models typically show an impressive level of accuracy while emphasizing on minimizing the false positive rate. However, with each proposed model iteration, we get further from the setting in which these models are intended to operate. As kernels become more ornate and hardened, the implementation space for anomaly detection models is narrowing. Furthermore, the rapid advancement of operating systems and the underlying complexity introduced dictate that the sometimes decades-old datasets have long been obsolete. In this paper, we attempt to bridge the gap between theoretical models and their intended application environments by examining the recent Linux kernel 5.7.0-rc1. In this setting, we examine the feasibility of syscall-based HIDS in modern operating systems and the constraints imposed on the HIDS developer. We discuss how recent advancements to the kernel have eliminated the previous syscall trace collect method of writing syscall table wrappers, and propose a new approach to generate data and place our detection model. Furthermore, we present the specific execution time and memory constraints that models must meet in order to be operable within their intended settings. Finally, we conclude with preliminary results from our model, which primarily show that in-kernel machine learning models are feasible, depending on their complexity.**

***Index Terms*—host-based intrusion detection, system calls, hidden Markov model**

I. Introduction

The problem of developing a reliable HIDS (or Host-based Intrusion Detection System) has vast implications as such a development would revolutionize the field of computer security. However, developing a detection model is challenging because both false positives and false negatives have consequences that would make the resulting HIDS impractical. Additionally, the landscape is largely dynamic as malware and computer systems continue to progress and evolve. Thus, the definition of normal computer behavior and anomalous computer behavior is constantly changing. Finally, for a HIDS to be feasible, the anomaly detection algorithm must meet certain resource consumption requirements so that the overhead introduced does not overly degrade the usability of the system.

The origin of the HIDS problem can be traced back to 1996 when Forrest et al. proposed a method to detect abnormal system behavior by building a model using sequences of system calls [1]. While this paper offered a foundation for future research, perhaps the most important discovery of this was an approach to measuring computer behavior. In fact, they showed that sequences of system calls for root-level processes provided a stable signature for normal system behavior. However, being the initial research into the field, their detection model was a simplistic database approach: if a transition was not represented in the database, then the said behavior was indicative of an anomaly. While their model was never implemented in a live system environment, their results suggested a sensitivity towards false positives.

Subsequent research has looked deeper into the issue of false positives, and new model architectures emerged. Probabilistic schemes, such as Hidden Markov Models (HMM), have become a standard for HIDS [2]. And many complex architectures, mirroring advancements in the machine learning community, are still emerging, such as multi-layered approaches [3], or approaches using current neural-network technologies [4].

While these models are showing improvements on the detection and false positive rates, they are incrementally abstracted from the environment in which they are intended to be used. A deep learning approach with a minuscule false positive rate, for example, has little value if the implementation is infeasible. Furthermore, many of these models use the ADFA-LD dataset (2013) [5], the DARPA dataset (1998) [6], or the UNM dataset (1996) [7], which are sorely outdated.

Since recent kernel hardening has disallowed programming the in-kernel syscall handling mechanisms, the classical method of collecting syscall traces (e.g., writing syscall table wrappers to store all syscalls requested) has become obsolete. As a result, data sources much of the current research is based upon, all date back to a time prior to these changes to the kernel. We investigate several different approaches to collecting modern syscall data, and, as one of the main contributions of this paper, we propose a modern method based on extended BSD Packet Filter (eBPF) [8]. This method will allow researchers to generate and maintain an up-to-date syscall dataset, which will capture both the complexity and evolving nature of modern operating systems. Furthermore, the same kernel hardening developments have made it challenging for developers to implement a syscall based detection model, as these models must reside within the in-kernel syscall handlers. As our second main contribution, we propose a method for implementing anomaly detection models and explore the constraints of our framework. Finally, we demonstrate the feasibility of our proposed framework by using a barebones, placeholder HMM model. Our findings show that the framework imposes certain memory requirements due to the environment in which it lives. Furthermore, the environment demands certain execution time requirements in order to provide a usable system. However, while these concerns may constrain what models we can implement, we show that they do not eliminate the possibility of a reliable model living within syscall handling methods.

In this paper, we attempt to bridge the gap between proposed anomaly detection models and the requirements for implementing such imposed by the constraints of their intended environment. Rather than focusing on building a better model in an abstract environment, we begin to investigate how the dynamic nature of operating systems and resource consumption constraints will shape the design of anomaly detection models. Based on this investigation, we propose a framework for implementing a HIDS in a modern operating system.

The rest of this paper is organized as follows. Section II summarizes related work on developing HIDS architectures, and the inherent limitations of such architectures based on architectural design choices. Section III describes the impact of kernel hardening on programming syscall handlers and the current state of different approaches. Section IV details our proposed framework. Section V provides information regarding implied resource constraints of our framework and the limitations of the model we developed for proof-of-concept purposes. Section VI presents the results of our approach, and Section VIII summarizes the contribution of our work and outlines directions for future research.

II. Related Work

Most research in the area of HIDS focuses on improving model accuracy. However, there are several studies which touch on aspects of the implementation and system design problem. Pendelton et al. investigate the problem of generating a modern syscall dataset for HIDS research [9]. However, they make use of the Pin technology, which limits the scope of applications to Intel x86 architectures, as the Pin technology is specific to that architecture [10]. Furthermore, Pendelton et al., and projects we have seen that are not based off the historical datasets, make use of `strace(1)` [11] to capture syscalls [12]. While this technology is well known and easy to use, it is limited in its abilities. The `strace(1)` function simply intercepts and returns a syscall, its arguments, and the related process ID. Unfortunately, it does not offer an interface wherein we can add our own programming logic or features to the trace extraction facility. Therefore, it is limited to only include the features that are built in, and consequently certain things (like filtering based on process permission levels – a requirement for HIDS) are not possible. Therefore, in these studies, we do not see any filtering on process privilege level. Additionally, due to the same limitation of `strace(1)`, there is an inherent delay between monitoring and detection. This is because the analysis of the syscall trace is not being done within the syscall handler itself, but rather in a different scope. Thus, even though methods can be used to limit the delay and develop a pseudo real-time system, HIDS based on this technology will only be able to detect intrusions shortly after they occur – which is not useful for prevention. We see this limitation in systems based on this approach or similar approaches [12], [13]. On the other hand, the framework we propose embeds the detection within the syscall handlers themselves, offering true real-time detection and a system which can prevent malicious activities from happening.

III. System Design Considerations

Of utmost importance to the implementation of our syscall based HIDS is the point of interaction between the userspace and the kernel in the process of servicing a syscall. All user syscall requests will need to go through this point of interaction, and the kernel will return from here back to the userspace. Thus, this represents an ideal location to gather syscall data, and, further, to analyze syscall behavior.

The development of our system is based on the most recent version of the Linux kernel (5.7.0-rc1). Based on this kernel, we have identified three different mechanisms to use for implementing a HIDS: (1) adding a feature to the kernel, (2) implementing a loadable kernel module, or (3) placing tracepoint event triggered code.

A. Kernel Feature

Due to the fact that all syscalls will pass through the syscall handler on entry and on exit, adding a feature to this aspect of the kernel is a natural choice to implement a HIDS. Given that our system has an x86_64 architecture, the relevant file is found within the Linux kernel tree's /arch/x86/entry directory as entry_64.S. To implement our system here, we would need to hardcode a trace extractor and our developed model. This implies at least two development cycles, as we would need to utilize the extracted trace for developing the model. Additionally, any minor tweaks to the model would require further hardcoding into the kernel. Further, the file is written in assembly, which is relatively inaccessible when compared with higher-level languages, such as C. Moreover, the kernel build process is long. It took us approximately 45 minutes when multithreading the build on eight cores of an i7-3770 @ 3.4GHz. Finally, we would need to repeat the process across all architectures to achieve portability. Thus, this environment is impractical for prototyping.

B. Loadable Kernel Module

We intended to use this approach at the onset of our implementation. To implement our system here, we would write a wrapper for interrupt service routines (ISRs) and have every entry within the syscall_table point to our wrapper (this process is well known as syscall hooking). Thus, whenever a syscall is requested, the index of the syscall_table would point to our wrapper, which does either analyzing or extraction and subsequently points to the original syscall. This approach has the added benefits of being easily portable, supporting on-demand loading, and having a higher-level language development environment.

However, we immediately recognized several factors that make this approach infeasible. First, the syscall_table symbol is not exported to the module development environment by default. While this can be fixed by adding code to ensure the symbol is exported, rebuilding the kernel with the necessary configurations makes implementing the HIDS across multiple systems much less practical. Furthermore, the syscall wrapping approach is common practice for rootkit developers, and, as such, the kernel has been hardened against programs that try to do such. Kernels that are later than version 2.6 keep the syscall_table in a write-protected area of memory to thwart these attacks. While there is a potential workaround by using write-unlocking kernel hacks, we prefer not to go against the grain of the intended functionality of the kernel as this will likely cause issues in temporal portability.

C. Kernel Hooks

Leveraging kernel hooks provides a feasible avenue for HIDS implementation. The basic premise is that we can "activate" hooks at various points within the kernel, by instructing the kernel to execute our various code whenever the kernel encounters the hook. Obviously, due to security considerations of such a feature, there are usually strict limitations as to what is allowed within the hook code. Kernel version 5.7.0-rc1 supports several different tracepoints, including (among others): perf_events, kprobes, and tracepoints [14]. Each of these tracepoints provides insight into kernel behavior by executing code at points of interest within the kernel.

Perf-events are a conglomerate set of events from many different categories of kernel hooks, such as uprobes (userspace dynamic tracing), kprobes, tracepoints, and sockets (network specific tracing). What makes perf-events unique is that they are a component of `perf(1)` [15], which is a complete system monitoring software. `perf(1)` provides a command-line interface from which users can sample data and print summary reports. While this is convenient for some tasks, the programmability is too limited in scope for building a HIDS.

Kprobes provides a *dynamic tracing* facility in that users can attach to any function within the kernel as they see fit. This provides a flexible set of events to monitor. However, the API is not guaranteed across kernel versions, and any minor update to the kernel could crash a kprobes-based system.

On the other end of the spectrum, the standard tracepoints are a feature of the kernel itself. As of kernel version 2.6, developers of the kernel included hooks into the kernel that can be leveraged by users. Since these are shipped with the kernel (referred to as *static tracing*), we can trust the stability of the API across updates. The cost of this, though, is that tracepoints are limited in scope in terms of which events are monitored (e.g., tracepoints can only activate hooks that were previously hardcoded into the kernel). However, the syscall_handler is the primary event that we are interested in (see Section IV) and is supported by tracepoints. Due to our primary design goal of security, coupled with the stability of tracepoints, we opt to use the tracepoints facility as our hooking mechanism.

D. extended BSD Packet Filter (eBPF)

While we have opted to use tracepoints as our avenue for development, we still want a high-level interface to these events (a system such as `perf`, but with more programmability). This is exactly what eBPF offers. BPF (BSD Packet Filter) was originally designed in 1992 as a way to instrument network events within a system [16]. However, in 2014, BPF was beginning to be expanded to eBPF, and, in 2017, the modern version of eBPF was included in kernels [8]. eBPF (which we will confusingly refer to as BPF as per community standard) extended the original BPF by allowing more programmability and a greater number of attach points (e.g., for arbitrary kernel events rather than just sockets). Thus, it offers a development environment for a much wider range of tools.

BPF is technically an in-kernel virtual machine [17]. The virtual machine has a userspace front-end, which is made possible through BCC (BPF Compiler Collection). However, we will refer to the system itself as BPF. To interact with BPF, a user can use the interface within a familiar front-end development environment, such as Python or C. The embedded BPF code within the file will be compiled down to BPF bytecode. Moreover, upon execution of the program, the bytecode will be passed through the verifier before landing in the BPF virtual machine. As a mechanism to provide security and stability of the kernel, the verifier currently disallows certain things. We will discuss these limitations in Section V. Importantly, we are able to specify which point in the kernel we want to attach our BPF virtual machine to. We can thus specify the syscall_handler tracepoint. Finally, the BPF system makes use of a concept called maps. These maps can be thought of as pre-defined composite datatypes offered to the programmer. The key feature of these is that we can build them in the userspace and reference them from within our code attached to the kernel hook. There are several different map structures, all of which offer persistent, in-kernel storage. Thus, we are able to store large amounts of data between tracepoint executions.

IV. SYSTEM DESIGN

With all preliminaries accounted for, we begin to build our system. As seen in Figure 1, there are three major pieces to the system – trace-collection, machine learning (ML) engine, and analyzer.

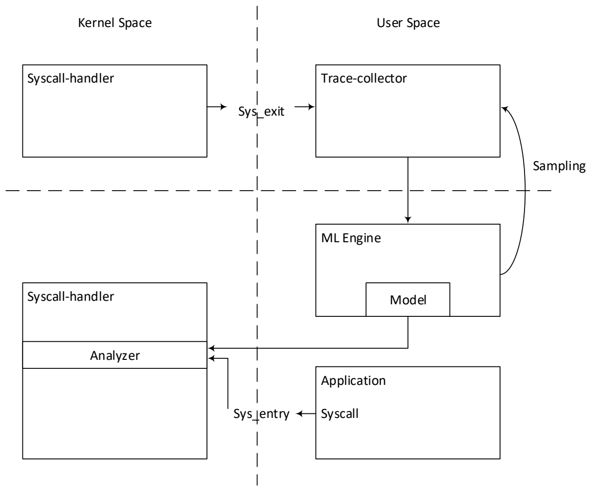

Fig. 1. System design

A. Trace Collection

In the sampling phase, we attach to the syscall-handler, specifically at the sys_exit tracepoint event, designating the exit point of the syscall handler. The reason that we attach to the exit point is that we are only interested in developing a trace of successfully completed syscalls in sequential order. By attaching to the exit point, we are less likely to include the incomplete syscalls. We store its thread-id, process-name, and syscall requested for each syscall corresponding to a root-level process. We drill down by thread-id for the case of multi-threaded processes: since sequentiality is of utmost concern for our trace, we do not want to shuffle together syscalls for multiple traces. There is some current research focused on the specific problem of generating data for syscall based HIDS [9]. The reason behind this research is two-fold: (1) much research in the field is based on three historical datasets [18] (published in 1996, 1998, and 2013, respectively), which are outdated and not-clearly labeled; (2) historically, if researchers wanted to develop their own dataset, then this would entail a lengthy process. However, leveraging BPF (specifically BPF-Trace), we were able to generate a dataset with one line of code. Considerations of multi-threading, process-trace creation points, data granularity, and sufficient sampling are left to a separate piece of work. For our present considerations, we traced the entirety of syscalls for root-level processes over five days. After completion, we had millions of data points representing GBs on our disk. This is sufficient data to build our ML engine.

B. The ML Engine

In the heart of the ML engine, we leverage existing algorithms in the Pomegranate library [19] to train an HMM. On either end of the ML engine, we implement wrappers to implement data pre- and post-processing concerns. There is a significant amount of research showing HMMs as a good choice for this problem as they can hold false negatives to a desired standard while keeping false positives at a minimal level (preliminary research dating back to 1999 shows promising results [20]) [2]. HMMs essentially boil down a system to a set of hidden states, each with a probability of emitting an observed data point (syscall) and a probability of transitioning from one hidden state to another (for an excellent treatment of HMMs see [21]). Thus, the parameters of the model are the aforementioned probability matrices, and a probability vector designating the probability that our system starts in a given hidden state.

During the training phase, a user specifies N as a hyperparameter designating the number of hidden states. While research has shown that an appropriate N is one that is close to the number of unique syscalls within a given trace [2], we chose a small N (for our first iteration, $N = 4$) for rapid prototyping reasons. The primary concern in this research is to illuminate storage and performance constraints for a real-time HIDS; thus, an easily implemented model is a logical choice. Additionally, we chose to build only one model (based on our longest trace – systemd with 1.4 million syscalls), while a complete HIDS requires one model per process. The reason why we chose to model one process follows the same logical vein as limiting the number of hidden states. After training our model, we extract the necessary parameters (i.e., the initial probability vector, transmission matrix, and emission matrix). To make these data available to our BPF program, we reference the maps the program uses from userspace to "upload" the model into the kernel. Since we developed the model using Python, and the kernel treats data in a C-like manner, we must first convert our data to have a C-type structure using Python's `Ctypes` library [22].

C. The Analyzer

In our current system, the analyzer is implemented in the exit point of the syscall handler. The reason we chose to attach here is that the observed trace at the exit point will likely conform closer to our collected trace (which was collected at the same tracepoint). Ideally, we would like to attach to the entry point as we could implement preventative measures here to block intrusions; however, implementing at this point in the kernel introduces variances in the data that we need to account for in our final model. For example, it is possible that an application re-requests a syscall because its original request failed. Our trace collector aimed to gather only successfully completed syscalls, so such re-requesting behavior, although an occurrence within normal behavior of a system, is not included in the data on which our model is based.

Since the analyzer runs within the BPF virtual machine, we need to develop our functionality with the verifier and kernel context in mind. The major concerns are (1) stack size allowed by the verifier and (2) lack of floating-point arithmetic within the kernel. Since we make heavy use of floats/doubles when handling probability calculations, we need to convert these operations to standard arithmetic operations. Thus, we need to scale up the probabilities by a constant factor to convert them to unsigned longs (ints). Additionally, since our stack space is limited to 512 bytes, we choose to represent our probabilities as 32-bit unsigned ints (each taking up 4 bytes). To scale up our probabilities, we multiply by a constant factor 10^N and maximize N s.t. all values can be represented by a 32 bit unsigned int. Thus, $N = floor(\log_{10}(2^{32} - 1)) = 9$. This implies a lack of precision, as some values determined by our training had precision extended beyond the 350th decimal place. Balancing precision and storage space is left as a problem for future research.

The BPF attach point for our analyzer is the exit point for the kernel syscall handler. Thus, every time it runs, our analyzer acts on a specific syscall for a given process. We use the forward algorithm [21] to determine the probabilities that this syscall occurred given a specific hidden state. This gives us our new trellis, which, when taken its sum, gives us the likelihood that this syscall occurred given our model of the system. We set the threshold probability to 0.001%, which, if broken, indicates an abnormal behavior. The threshold probability provides a mechanism to balance false positives against false negatives. The fine-tuning of such is left to future iterations of this system. Finally, since we write back our trellis to be used for the next syscall, each time a syscall occurs, we evaluate a syscall within the context of the already occurred trace. However, since $p^n \to 0$ as $n \to inf$, for $0 < p < 1$ and n beginning at an arbitrary non-negative integer, our probabilities will always break the threshold as our trace gets arbitrarily long. Thus, in future iterations, we will need to control the length of the trace analyzed.

V. Restrictions of this approach

The end result of using the framework we outlined in the previous section is to place code within kernel's syscall_handlers. Thus, the interface we used imposes restrictions on programs to ensure the design goals of the kernel (e.g., performance and security related goals). As such, certain features available to standard C programs are not available to BPF programs. Perhaps the most notable restriction is the stack size. At the time of this writing, the stack size for BPF programs is limited to 512 bytes. Thus, the designers of the analyzer system will have to make extensive use of maps to hold model parameters. These structures offer essentially limitless storage for practical purposes. If a larger stack is required, BPF allows developers to chain together multiple BPF programs. Therefore, designers can effectively increase the stack size available to their algorithms. Nevertheless, chaining together multiple BPF program introduces new overhead costs. Additionally, care must still be given to program complexity as execution time is a significant concern due to the intended environment.

Another significant restriction is that there is no floating-point support. The developers must design a clever system that balances accuracy and the size of local variables. A less significant but still notable restriction concerns the runtime of programs: storage and loop requirements must be determined by the compiler. That is, we can not dynamically allocate memory, and we can not have infinite loops. The interested reader may consult [14] for a full list. While developing the analyzer system, researchers must be aware of these constraints in order to attain program legality using the framework we outline.

The analyzer system was developed in accordance with the above restrictions. As a result, we were able to develop a legitimate HIDS within the kernel of the most recent version of the Linux kernel. However, the model was intended to be a placeholder as a proof-of-concept mechanism for our framework. Thus, the model itself has severe limitations, and we are not proposing it as an actual anomaly detection model. We direct the interested reader to [2] to see state-of-the-art models designed around optimizing accuracy metrics. It is left as a focus of future research to build these models in accordance with our framework to make them legal models.

For the time being though, our model only contains three hidden states. This is a severe limitation, as prior research has shown that the number of hidden states should be roughly equal to the number of distinct syscalls within a process' trace [2]. We made this design choice because we wanted to perform a preliminary analysis on the constraints imposed by our framework; thus, we were more concerned with generating a model with a similar structure to the proposed models rather than one containing all the prediction features of a proposed model. Nevertheless, since our model contains only three hidden states (compared to the 40 seen in the traces built in our data collection phase), our model will contain significantly fewer features that correspond to a system's normal behaviors. To estimate the scale, we recognize that our model contains three states. Thus, our transition matrix contains only nine entries, each of which representing the probability of transitioning from one state to another. Considering the finding of prior research, that a suitable number of hidden states is around the number of distinct syscalls present in a process' trace, we see that we would ideally like to give a probability measure for 160 transitions. In such a model, we can safely assume that each syscall present in the trace will be represented significantly by at least one state. Therefore, the transition matrix contains all necessary information to capture all the transition features in our sequence data. However, since we compress the transition matrix to 5% of the size, many transitions are not given enough, if any, probability of occurring, and the accuracy results of our model will be consequently diminished.

VI. Results

The primary focus of this research is on space requirements and latency in a real-time environment. There are two main components of space requirements: memory required to hold our model, and minimum stack size required to build our in-kernel analyzer for a given model. Given that our framework restricts the stack size we have available, we are chiefly concerned with the implied stack size given an HMM. In the design of our implementation of the analyzer, rather than using references to memory provided by the maps, we decided to create local variables to hold the parameters of our model. This is an execution time-based performance decision and is obviously not the only way to design an analyzer. That said, we will look at the size of the parameters required by the analyzer to estimate the stack size implied by our model. This estimate will strictly be a lower-bound because the stack will include things other than the parameters (such as other local variables and calling convention related storage). However, this estimate will provide a relatively tight lower-bound as the data that uses most of the stack space will be the parameters of our HMM model. Our analyzer will need three sets of parameters. However, since our analyzer will be called to analyze a given syscall for a given process, we will only need a very small subset of the parameters at each invocation of our analyzer. In fact, we will need ($N \times N \times 4$) bytes to hold the transmission probabilities, and ($N \times 4$) bytes for the emission probabilities and an additional ($N \times 4$) bytes for the trellis structure. Thus, for our 3-state model, the parameters required 60 bytes of stack space. If we intend to use local variables for our parameters, these findings imply an upper limit of 10 states before we need to chain together BPF programs to appease stack limit constraints. Execution time and stack space design considerations for more robust models are left to future research.

Another important consideration is the amount of additional memory the kernel requires as a result of our HIDS. To analyze the total memory consumed, we will look at our maps. Our analyzer required three separate maps to hold the model and the current iteration's probabilities. The transmission matrix map held values that used ($N \times N$) $\times$ 4 bytes (where N is the number of hidden states) and were indexed using a 16-byte key. The emission matrix map held values that used $N \times 4$ bytes, and used a 20-byte key (16 to index the process and 4 to index the syscall). For each of these maps, we would need an entry for every process being modeled. Finally, the trellis map held values that used $N \times 4$ bytes and used a 4-byte key (relating to the thread ID of the process). We would need to have an entry for every thread. For theorizing, we assume the average number of threads per process in a live environment is 2 - this seems realistic as a majority of processes have one thread, with significantly less having 2-4, and a very small amount of processes (such as Internet browsers) having many threads. Thus, the equation to determine memory requirements for p models (p representing the number of processes) with N hidden states is (36 + $4N(N + 1)$) $\times$ p + (4 (N + 1)) $\times$ $2p$ bytes.

For our model with four hidden states and one process, the memory requirements were roughly 156 bytes.

Additionally, we built a tool that measures the elapsed time between a syscall exit and a syscall entry. Using this tool, we are able to infer the execution time for handling a syscall. Using an i7-3300 @ 3.4 GHz processor, we collected syscall execution times for the Linux 5.7.0-rc1 kernel over the course of 30 minutes, which led to a latency dataset that was on the scale of tens of millions of data points. We ran our HIDS in the same time-frame and measured the execution time of the analyzer embedded into the syscall handler.

TABLE I
ANALYZER OVERHEAD ANALYSIS - SYSTEMD

SYSCALL	Execution Time (ns)	Overhead (ns)	Execution Time Increase	# of syscall observed
openat	330442	1933	0.59%	691
close	48484	1533	3.16%	599
fstat	41302	1603	3.95%	489
kcmp	30419	876	2.88%	321
read	1252304	1999	0.16%	194
epoll_wait	314323056	17229	0.01%	162
recvmsg	2902915	3734	0.13%	142
newfstatat	592	2020	341.40%	114
getdents	226837	1783	0.79%	70
timerfd_settime	714	2377	332.82%	56

Table I encompasses the average latency of handling the syscall and the average execution time of our analyzer for that same syscall for the top 10 most frequent syscalls in our run. Our analyzer predicted against 3093 syscalls in this time frame, with roughly 20% of syscalls being *openat*. We see that the geometric mean for the execution time increase of these syscalls, which measures the amount of overhead our analyzer requires in terms of the original execution time of the given syscall's handler, is 17.35%. Since the analyzer performs roughly the same calculations for every syscall, we observe that the raw execution time of the overhead introduced has a low variance, with a mean (weighted for the frequency of syscalls) of 2630 nanoseconds across all syscalls. Additionally, we observe that some of the most frequent syscalls analyzed had execution times on the scale of hundreds of thousands of nanoseconds. This, coupled with the observation that there was no perceivable difference in the operating system's execution, leads us to conclude that the introduction of our analyzer did not degrade the performance of the kernel in terms of execution time. It is worth mentioning that the prediction algorithm, which is based on the forward algorithm, is $O(N^2)$. Presumably, a 10-fold increase in the size of hidden states will result in roughly a 100-fold increase in execution time. This would put the overhead of the analyzer in the scale of hundreds of thousands of nanoseconds, which is comparable to the execution time of syscall handlers for a majority of the more frequently analyzed syscalls in our data. Thus, an analyzer based on an HMM with 30 hidden states is not infeasible due to execution time overhead.

Finally, we turn to the modeling results - an area that is explored in depth by the large majority of research on this subject. At every opportunity, we opted for ease of implementation rather than modeling fit. Due to this, we experienced poor modeling results.

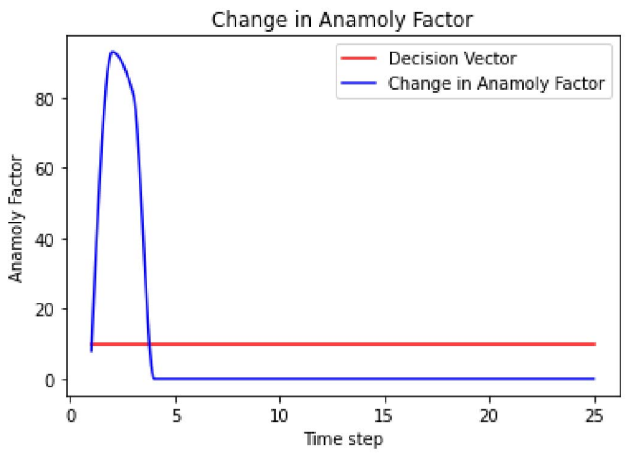

Fig. 2. Live anomalous state prediction - systemd

In most of our trial runs, we encountered false positives almost immediately. In Figure 2, we see that our analyzer initially calculated reasonable values for the anomaly factor. However, after a short period of time, our anomaly factor dipped below the decision vector (indicating we should flag the state of the system as anomalous), and converged to zero. This behavior can be simply explained. Within a given hidden state, the sum of all the probabilities of syscalls must be 100%. Additionally, each hidden state attempts to pick up a dominant feature of the data which usually consists of a string of 1-3 syscalls. Thus, we see that each hidden state assigns a proportionally large amount of probability to one or a few syscalls, with the remainder of the syscalls receiving very little, if any, probability. Thus, when a process exhibits behavior that is normal, but not within its most common behavior, the probabilities in our model drop drastically, and our system flags the behavior anomalous. Our model is a classical example of underfitting the data – which was done by design. Our design approach mandates that future research should be oriented on building a model for accuracy within the developed framework. A cursory analysis of our prediction results also suggests that more work is to be done on fitting before we can put forward a proposed mode. Thus a rigorous treatment on the confusion matrix results is not necessary.

VII. Discussion

The obvious next step is to attempt to implement an HMM, which better represents normal system behavior. It is clear that a more robust analyzer is needed to develop a feasible HIDS. Prior research has shown that using a number of states equivalent to the number of distinct syscalls a process uses provides good results. However, we have found that stack size restrictions impose an upper limit of 10 hidden states for an HMM. Furthermore, increasing accuracy for our parameters will impose an even more restrictive number of hidden states. We can work around this by either using memory references to retrieve the parameters or chaining together multiple BPF programs. However, both of these will degrade the execution time performance. Future research will examine the tradeoffs between model accuracy and execution time performance.

Additionally, using the formula derived in Section VI, we can easily determine whether or not such a model will be over-inhibitive from a memory performance standpoint. For example, we can determine that a 20-state HMM would require 376 KB memory assuming 200 processes - a realistic assumption based on our trace sample. This is a minuscule amount compared to the 1 GB of memory already consumed by the kernel base.

We began experimenting with the 20-hidden state model by training on our systemd process. This process has a trace length of roughly 1.4 million syscalls, which is much larger than any other trace. That said, training the model on this process was prohibitive, and, as a result, we do not report on the model here. During model training, 4 GB of RAM was consistently used for the running process with 18 GB used for cache storage. Additionally, 100% of a CPU's core was used. The training was done in a single-threaded manner using Pomegranate's implementation of Baum-Welch [19]. We expect a production system to learn in an adaptive manner, wherein the HIDS receives information from the userspace about false positives and gets incrementally "smarter" through this feedback mechanism. Thus a system will have onboarding learning phases. We can outsource this computation time to dedicated servers, which will continue to store incrementally smarter models. However, training is still prohibitive. Thus, an area to investigate involves speeding up the HMM training algorithms. In addition to implementing a parallel Baum-Welch, the gradient descent algorithms that have brought neural network models to the limelight are a potential source of improvement.

A final consideration is the fact that HIDS themselves are vulnerable to attack. Due to the fact that the detector makes predictions based on features of the data, attackers can manipulate their malware in such a way that the syscall trace emitted by their program exhibits features that represent normal system behavior [23]. To thwart this mimicry attack, the developers of the detector system can include more features in their model via either including more states (if using HMM), by including the arguments to syscalls, or both. Much research has focused on improving the detection mechanism of a HIDS, and the interested reader is directed to [2]. While we have given evidence that suggests larger models are feasible within our framework, we have not directly implemented a larger, more realistic model. Due to the fact that the runtime for training HMM was prohibitive to the scope of our work, we leave that as work for a future project.

VIII. Conclusion

In this study, we developed a framework that bridges the gap between research on improving accuracy for anomaly detection models and developing a HIDS for a modern operating system. Along the way, we demonstrated a method for generating new-age syscall sequence datasets on the fly. We pointed to a few concerns that need to be addressed before a full-fledged dataset is published.

Additionally, through our implementation of a naïve HMM, we were able to estimate the storage and processing requirements for implementing a real-time HIDS. When addressing program size concerns, we found that HMM models that contain up to 10 hidden states can be implemented with no penalty to the execution time of the analyzer. Using a different combination of design choices, we could increase the number of hidden states, but this will likely come at the expense of performance in terms of execution time.

We found that the overall memory requirement for our HIDS is on the scale of $O(N^2)$, where N represents the number of hidden states in the model. We developed a formula which can accurately calculate the memory requirements and determined that, for a model with 20 states, our HIDS will consume roughly 376 KB of memory. Furthermore, if we double our number of states, we can expect a four-fold increase in memory requirements, Thus we expect a model with 40 hidden states to require about 1.5 MB of memory. Neither of these values is prohibitive when considering the memory requirements of the kernel.

Finally, we also found that the overhead our system introduces on syscall handlers is $O(N^2)$. Based on empirical analysis and extrapolation, we informally conclude that a HIDS with 30 states will not be infeasible in terms of the overhead introduced to syscall handlers. However, with 30 hidden states, we effectively double the execution time of the majority of frequently used syscalls in our experiment. Thus, if we were to continue to increase the number of hidden states beyond 30, the predictor would begin to be the driving force of execution time for syscall handlers.

References

[1] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *Proceedings 1996 IEEE Symposium on Security and Privacy*. IEEE, 1996, pp. 120–128.

[2] J. Hu, "Host-based anomaly intrusion detection," in *Handbook of information and communication security*. Springer, 2010, pp. 235–255.

[3] X. D. Hoang, J. Hu, and P. Bertok, "A multi-layer model for anomaly intrusion detection using program sequences of system calls," in *Proc. 11th IEEE Int'l. Conf.* Citeseer, 2003.

[4] A. Chawla, B. Lee, S. Fallon, and P. Jacob, "Host based intrusion detection system with combined CNN/RNN model," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2018, pp. 149–158.

[5] 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.

[6] R. K. Cunningham, R. P. Lippmann, D. J. Fried, S. L. Garfinkel, I. Graf, K. R. Kendall, S. E. Webster, D. Wyschogrod, and M. A. Zissman, "Evaluating intrusion detection systems without attacking your friends: The 1998 DARPA intrusion detection evaluation," MIT Lexington Lincoln Lab, Tech. Rep., 1999.

[7] "Sequenced-based intrusion detection," https://www.cs.unm.edu/~immsec/systemcalls.htm, Last accessed 30 Sep 2020.

[8] M. Fleming, "A thorough introduction to eBPF," *Linux Weekly News*, 2017.

[9] M. Pendleton and S. Xu, "A dataset generator for next generation system call host intrusion detection systems," in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*. IEEE, 2017, pp. 231–236.

[10] "Pin — a dynamic binary instrumentation tool," https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, Last accessed 10 Nov 2020.

[11] "strace(1) — Linux man page," https://man7.org/linux/man-pages/man1/strace.1.html, Last accessed 10 Nov 2020.

[12] A. S. Abed, C. Clancy, and D. S. Levy, "Intrusion detection system for applications using Linux containers," in *International Workshop on Security and Trust Management*. Springer, 2015, pp. 123–135.

[13] M. Kadar, S. Tverdyshev, and G. Fohler, "System calls instrumentation for intrusion detection in embedded mixed-criticality systems," in *4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems (CERTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[14] B. Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[15] "perf(1) — Linux manual page," https://man7.org/linux/man-pages/man1/perf.1.html, Last accessed 30 Sep 2020.

[16] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture." in *USENIX winter*, vol. 46, 1993.

[17] B. Gregg, "Linux extended BPF (eBPF) tracing tools," 2018.

[18] R. A. Bridges, T. R. Glass-Vanderlan, M. D. Iannacone, M. S. Vincent, and Q. Chen, "A survey of intrusion detection systems leveraging host data," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–35, 2019.

[19] "pomegranate," https://pomegranate.readthedocs.io/, Last accessed 3 Oct 2020.

[20] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)*. IEEE, 1999, pp. 133–145.

[21] D. Jurafsky and J. H. Martin, *Speech & language processing*. Pearson Education India, 2019.

[22] "ctypes — a foreign function library for Python," https://docs.python.org/3/library/ctypes.html, Last accessed 3 Oct 2020.

[23] 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*, 2002, pp. 255–264.