Implementing Large-Scale Autonomic Server Monitoring Using Process Query Systems

Christopher Roblee Vincent Berk George Cybenko
Institute for Security Technology Studies
Thayer School of Engineering
Dartmouth College, Hanover, NH 03755
Firstname.Lastname@Dartmouth.EDU

Abstract

In this paper we present a new server monitoring method based on a new and powerful approach to dynamic data analysis: Process Query Systems (PQS). PQS enables user-space monitoring of servers and, by using advanced behavioral models, makes accurate and fast decisions regarding server and service state. Data to support state estimation come from multiple sensor feeds located within a server network. By post-processing a system's state estimates, it becomes possible to identify, isolate and/or restart anomalous systems, thus avoiding cross-infection or prolonging performance degradation. The PQS system we use is a generic process detection software platform. It builds on the wide variety of system-level information that past autonomic computing research has studied by implementing a highly flexible, scalable and efficient process-based analytic engine for turning raw system information into actionable system and service state estimates. [1]

1 Introduction

Proper security and performance monitoring of modern server farms is a difficult task. While most commercial monitoring solutions do scale to the capacities required for capturing large amounts of status data being generated by high-throughput server banks, they are weak on the analytic capabilities required to interpret or otherwise use that data in such a way as to implement autonomic behaviors. For example, intrusion detection software often generates hundreds of alerts per minute, most of which are false-positives, subsequently desensitizing administrators to any actual serious threat. Additionally, it is not uncommon for most servers to share identical software, making the entire server bank vulnerable to the same set of attacks. When one server is compromised but not quarantined immediately, the attack may spread to the entire server array. The results can be catastrophic; unplanned downtime can cost as much as $550,000 per hour in lost revenue for large server farms, such as those used in the financial services industry [12].

Autonomic computing approaches to server management strive to identify deviant behavior by services running on the servers as a means of mitigating this problem. By having each server closely monitor all of its processes and system variables, it is possible to identify rogue or anomalous behavior when it occurs. Once identified, the process can then be quarantined, shut down, and possibly restarted. One major problem with modern autonomic monitoring approaches, however, is the way in which they monitor processes. They either use a limited number of status indicators or are invasive with respect to the operating system or application software. Some experimental systems, for instance, hook into the OS kernel and monitor all the system calls coming from a monitored process [7]. When combined with dynamic behavior learning algorithms, such methods can impose an unacceptable load on system resources.

In this paper we present a significantly more scalable approach to monitoring system processes. Process Query System (PQS) [4] technology allows us to quickly and easily integrate multiple sensor sources and model human system administrator analysis in order to obtain fast and accurate results, with a significantly reduced overhead as compared with other approaches. Human system administrators often only monitor a small portion of the available system data, such as server load, network load, I/O load, and intrusion detection data. By combining this information, a human analyst may decide to further investigate the behavior of any one server. Process Query Systems is a technology that allows quick and easy integration of sensor resources and

[1] This work was supported by: ARDA P2INGS Award No. F30602-03-C-0248, and DHS Award Nos. A3N308035 and 2000-DT-CX-K001. Points of view in this document are those of the author(s) and do not necessarily represent the official position of ARDA or DHS.

uses custom built models to detect evidence of occurring processes in the observed environment. Because of its level of abstraction, the programmer can focus on these process models instead of system design or sensor input. By correlating externalized host state with network-level context, PQS models can provide administrators with a clearer picture of network state and facilitate self-healing actions.

Our system combines user-space process monitoring, such as system load, I/O load, and fork behavior with intrusion detection data and loads it into the PQS processing core. We will show how our process models can significantly reduce false positive rate of the connected IDS systems, and how deviant process behavior can be easily detected, and qualified without a significant increase in server system load.

The next section contains a short introduction to the concepts behind Process Query Systems. Section 3 describes the server monitoring architecture we have implemented using PQS, including the sensor types and process models. Experimental results using a testbed server network are in Section 4 while Section 5 is an analysis of our findings with suggestions for future work.

2 Background

In this section we take an in-depth look at the new Process Query System (PQS) technology, as well as the current state-of-the-art in server monitoring systems. The particular implementation of a PQS that we used for this work was called "TRAFEN", the TRacking And Fusion ENgine [1]. We also discuss the general problem of server monitoring in more detail.

2.1 Process Query Systems

Process Query Systems are a new paradigm in which user queries are expressed as process descriptions. This allows a PQS to solve large and complex information retrieval problems in dynamic, continually changing environments where sensor input is often unreliable. The system can take input from arbitrary sensors and then form hypotheses regarding the observed environment, based on the process queries given by the user. Figure 1 shows a simple example of such a model. Model $\mathcal{M}_1$ represents a state machine $S_1 = (Q_1, \Sigma_1, \delta_1)$, where the set of states $Q_1 = \{A, B, C\}$, the set of observable events $\Sigma_1 = \{a, b, c\}$, and the set of possible associations $\delta_1 : Q_1 \times \Sigma_1$ consists of $\delta_1 = \{\{A, a\}, \{B, a\}, \{B, b\}, \{C, c\}\}$. Notice how this process is able to produce observed event a in both state A and state B. A possible event sequence recognized by this model would be:

$$e_1 = a, e_2 = a, e_3 = b, e_4 = c, e_5 = b$$

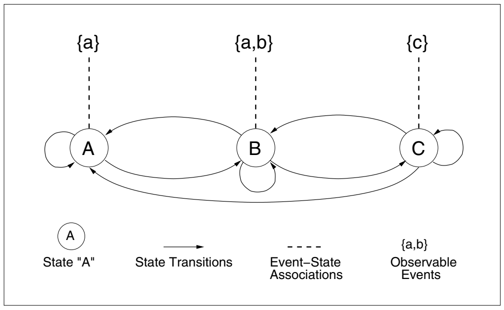


Figure 1. A Simple Process Model, $\mathcal{M}_1$

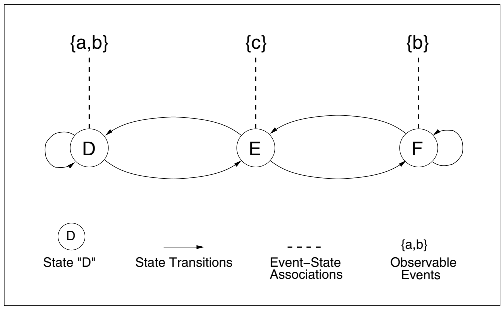


Figure 2. Another Process Model, $\mathcal{M}_2$

which we will write as $e_{1:5} = aabcb$ for convenience. Possible state sequences that match this sequence of observed events could be $AABCB$, or $ABBCB$, both of which are equally likely given $\mathcal{M}_1$. A rule-based model would require many rules to identify this process, based on all the possible event sequences. Below is a set of all the rules necessary for detecting single transitions:

$$\begin{array}{ll} AA \rightarrow & \{aa\} \\ AB \rightarrow & \{aa\}, \{ab\} \\ BB \rightarrow & \{aa\}, \{ab\}, \{ba\}, \{bb\} \\ BA \rightarrow & \{aa\}, \{ba\} \\ BC \rightarrow & \{ac\}, \{bc\} \\ CC \rightarrow & \{cc\} \\ CB \rightarrow & \{ca\}, \{cb\} \\ CA \rightarrow & \{ca\} \end{array}$$

Needless to say the list of possible event sequences for double transitions is massive (e.g. transitions AAA, AAB, ABB, ABC, ... etc.) and only grows when we consider dealing with the possibility of missed observations. Rules would then have to include sequences that have two transitions for a single observed event, albeit with a lower priority accounting for the fact that we expect only few observations to go missing.

We now introduce a second model $\mathcal{M}_2$, shown in Figure 2, defined by state machine $S_2 = (Q_2, \Sigma_2, \delta_2)$. Consider that both processes are regularly and concurrently occurring in the observed environment. Note that the process states are labeled differently, i.e. $Q_1 \cap Q_2 = \emptyset$, although both processes produce the same set of observable events, i.e. $\Sigma_1 \cap \Sigma_2 \equiv \Sigma_1 \equiv \Sigma_2$.

Now consider the following sequence of events:

$$e_{1:24} = abaacabbacabacccabacabbc$$

where each observation may have been produced by instances of model $\mathcal{M}_1$, model $\mathcal{M}_2$, or be totally unrelated. It must be noted that any modeled process may very well be occurring multiple times concurrently. A Process Query System uses multiple hypothesis, multiple model techniques to disambiguate observed events and associate them with a "best fit" description of which processes are occurring and what state they are in. By comparison, a rule-based system would become extremely complex for the above situation and quickly become unmaintainable and unintelligible. Additional problems with rule-based approaches arise when probabilities are assigned to the state transitions, and/or the event productions, as with Hidden Markov Models.

Consider the following example. Assume that the first model describes the dynamics of a propeller plane, and the second model describes the dynamics of a fighter jet, both observed by radar. It may very well be possible that there are several propeller planes and a group of fighter jets in the same airspace, all within radar range. The PQS will use the radar data as input observations together with the two models to disambiguate which radar observations were triggered by which aircraft by associating radar observations using the models. Subsequently, the hypothesis will be that there are several instances of the model $\mathcal{M}_1$ (the propeller plane) and a group of instances of model $\mathcal{M}_2$ in the observed environment. Since the environment is dynamic, the *top hypothesis* will be changing continuously as planes move in and out of radar range.

A PQS is a very general and flexible core that can be applied to many different fields. The only elements that change between different applications of a PQS are the format of the incoming observation stream(s) and the submitted model(s). Compare this with a traditional DBMS; inventory tracking systems, accounting, customer databases, etc. are all different applications that, at the core, are based on the same DBMS. Likewise we have implemented vehicle tracking systems, server farm monitoring applications, enterprise network security trackers, and covert timing channel detectors using the same PQS software core by simply supplying a different observation stream and a different set of models. Internally, a PQS has four major components that are linked in the following order:

1. Incoming observation handling and sensor subscription.
2. Multiple hypothesis generation.
3. Hypothesis evaluation by the models.
4. Selection, pruning, and publication.

To summarize, the major benefits of a PQS are its superior data management and applicability. The application programmer simply connects the input event streams and then concentrates on writing process models. Models can be constructed as state machines (above), formal language descriptions, Hidden Markov Models, kinematic descriptions, or a set of rules. All of these different model types are first compiled down to the fundamental PQML (*Process Query Modeling Language*) representation and then submitted to the PQS. The PQS is then ready to track processes occurring in a dynamic environment and continuously present *the best possible explanation of the observed events* to the user.

2.2 Related Work

In recent years there has been significant research into different implementations of self-awareness and self-healing in server environments. Dudley et al. describe the motivations and architectural concepts underlying much of the work in this space [5].

Commercially-available server monitoring platforms, such as NimSoft's NimBUS [11] and JJ Labs' WatchTower [10], offer robust, lightweight sensing and reporting capabilities across large server farms. However, these types of solutions are oriented towards massive data collection and performance reporting, and leave much of the final analysis and decision-making to the administrator. Our approach automates this analysis by identifying failure states probabilistically, based on behavioral models. Sottile et al. [13] and Ferreto et al. [6] also present scalable, lightweight architectures for cluster monitoring.

Kreidl et al. apply the host-based Autonomic Defense System to solve a similar problem through model-based detection and response [9]. Their approach involves offline training of Markov models to represent different attack scenarios. The architecture is different from ours, in that sensing, detection and control components run on the individual hosts themselves. We hope to use PQS to determine the viability of a joint network and host-based paradigm built on a centralized architecture for detecting and responding to server failures.

Theriault et al. compare local and orchestrated mechanisms for the autonomic detection and mitigation of a distributed e-mail denial of service attack [14]. Like our system, theirs uses data from multiple observation spaces to support coordinated detection of attacks.

Forrest et al. have proposed methods of detecting anomalous host behaviors by monitoring system call sequences of selected Unix processes [7]. This requires the offline construction of normal pattern databases for each monitored application. The online detection of anomalous traces is a difficult task to scale, however, as every server can exhibit different system call behaviors for the same applications. Runtime analysis of system calls can furthermore curtail the monitored server's performance.

Backdoors [2] is a system architecture for the remote detection and repair of OS damage with non-intrusive probes. This system imposes no overhead on OS resources, but requires the installation of a dedicated hardware control channel and specialized OS extensions (kernel hooks) to monitor performance metrics. Our platform uses passive, platform-agnostic sensors that run in user-space to extract similar OS state information. This sensing architecture is light-weight and portable, as it does not require any significant host re-instrumentation.

2.3 Problem Considerations

There are many conceptual and operational challenges in implementing a scalable and effective server monitoring framework. It is useful to first redefine the issue of scalability in the context of process query systems and server awareness.

We consider scalability at multiple levels in a monitored system as consisting of the following qualities:

1. Detection performance (e.g., false positives and false negatives).
2. Runtime performance of analytic engine (e.g., the TRAFEN core).
3. Sensor performance (e.g., quality of generated observable).

In order for our system to be scalable, these qualities should not degrade with increases in the following parameters:

1. Total number of monitored hosts.
2. Total number and complexity of monitored services per host.
3. Size of real-time generated hypothesis sets.
4. Number of submitted process models in the system (i.e., distinct scenarios tracked on a given node).

In general, as the number of tracked process instances grows, so does the difficulty of the decision making. In many ways, server monitoring is more complex than tracking physical objects, as the observables are significantly more interleaved and generally less reliable. For instance, in tracking a projectile or a vehicle, the observation space is limited to a set of measured coordinates, which have a well-defined meaning in the problem domain. When tracking multiple instances of related and unrelated server failures, however, one must deal with significantly more complex and less deterministic observables, such as resource utilization and execution behaviors.

One simple, yet significant advantage in monitoring servers over physical objects is the data and hypothesis reduction offered by deterministic host identification: unique IP addresses. We can easily use the servers' IP addresses as unique keys to map observations in the analytic engine. Therefore, detection performance should scale linearly with the number of monitored hosts, since we don't need to disambiguate between their emitted observables.

The true challenge thus becomes realizing generality in the decision logic with respect to each monitored node. If the models or sensors are over-trained for specific servers, then the logic will not scale well to networks of servers.

Another important question is whether or not there is a significant benefit in aggregating data from multiple hosts. This would potentially enable the learning of network baselines and the identification of outlier hosts. Considering the heterogeneity of most server environments, however, this could prove difficult.

3 Architecture

In this section, we describe the fundamental components of our approach to autonomic server monitoring, and how they integrate with our current PQS implementation, TRAFEN (TRacking And Fusion ENgine). The host-level component is a user-space sensor, which monitors host behavior and publishes significant events to a TRAFEN engine as observations. The TRAFEN engine fuses this information with observations from other sensors using custom process models, which hold the high-level evaluation logic. Models correlate host-generated events with events from other spaces, such as network sensors, that pertain to network-level indicators of host failure. From the output of these process models, administrators or higher-level TRAFEN instances can make valuable conclusions about the state of one or more nodes in the monitored network.

3.1 User-space Sensors

The purpose of the user-space sensor is to collect and pre-process host-level state data and to generate sufficient, meaningful events in order to enable host verification at higher levels in the system. The principal design consideration for these sensors is that they operate in a relatively non-intrusive manner, and that they require minimal host reinstrumentation upon deployment. Events are generated on a reactive basis when short-term or long-term anomalies in any of the monitored host metrics are detected. These events are subsequently published to the TRAFEN system

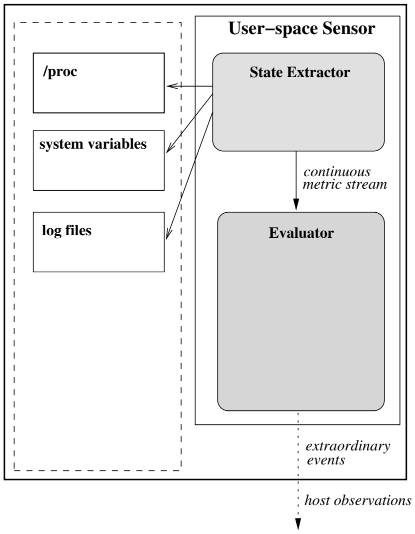


Figure 3. The user-space sensor running on monitored host.

as observations over a dedicated TCP socket connection. Observations are treated as possible symptoms of larger-scale host or service failures, or security compromises.

The user-space sensor, shown in Figure 3 comprises two decoupled modules: a state extractor and an evaluator. The state extractor samples a set of raw process-level and system-level performance metrics at regular intervals. The state extractor can be configured to monitor or exclude from monitoring any subset of processes. This allows for the reduction in the number of generated events, and subsequently noise, by limiting evaluation to specific applications. The state extractor's performance may also be tuned by adjusting its sampling rates. At present, this is the only machine-specific configuration required by the sensors. The state extractor acquires the following metrics:

For each monitored process:

- process ID (PID)
- process state
- percent memory utilization in MB
- percent process CPU utilization
- process parent name
- process parent ID (PPID)

For the entire system:

- percent total system memory utilization
- percent total system CPU utilization
- total number of processes
- total MB sent and received over network interface
- total number of disk IO operations
- total number of active sockets open

The raw data is extracted from the */proc* file system, while other system-level properties, such as IP address and time stamps are acquired through external calls to shell commands. These indirect measures of host behavior provide a fairly platform-agnostic view of a system's runtime state and can be extracted non-invasively, without reinstrumenting the kernel or application code. The evaluator consumes this continuous stream of state samples, and generates an event when it detects something extraordinary across them. The sensor provides for two basic event models:

- Routine, continuous events (e.g., *system CPU usage up 30%*).
- Relatively infrequent, discrete events (e.g., *httpd process forks a shell process*).

Both alert categories are highly valuable as potential manifestations of system or service failure to the detectors at the TRAFEN level, so we account for both in our process models and sensor architecture.

A fundamental challenge to developing robust host sensors for effective and lightweight anomaly monitoring is that many instances will in practice generate excessive amounts of redundant or insignificant events, possibly degrading host, network, and detector performance. It is often difficult at the host level to distinguish normal or acceptable variations in runtime variables from true anomalies. Many host sensors address this by capturing characterization baselines of each performance metric during an initial training phase. This can be a lengthy and burdensome process, however, and we hope to minimize the degree of machine-specific configuration. Extensive sensor training can reduce the generation of seemingly irrelevant observables (e.g., system CPU utilization increase). However, we do not want the evaluator to simply ignore these traces, as they might be valuable in a higher-level context as symptoms.

To manage the evident tradeoffs between monitor runtime performance, portability and detection accuracy, the evaluator aggregates performance metrics over sampling windows to effectively maintain smoothed, running averages. The sensor, in turn, listens for short-term and long-term deviations in OS and target application behavior by comparing each metric slope to threshold values. When a new sample exceeds the previous window average by this threshold, an event is generated. Hence, the slope thresholds and sampling window length control the sensitivity of the sensor. This coarser-grained analysis reduces the overall frequency of false alert generation and doesn't require *a priori* determination of acceptable behavior ranges.

3.2 Event Format

When a user-space sensor detects an anomalous trace in one of the monitored host metrics, it generates an event to describe the perceived host state and publishes it to TRAFEN. Each event is assembled as a TRAFEN observation - a structured, canonical format that only contains information that is relevant to higher-level entities. To enforce this level of granularity, significant host events are generally translated into higher-level semantics (e.g., *System memory up 15% → SYS.MEM*) . Observations are collections of name-value pairs and adhere to a simple ontology, demonstrated in the examples below.

```
HostObservation {
    time: Thu Jan 06 13:34:05 EST 2005
    type: SYS.PROCS.GROWTH
    ip: 10.0.0.20
    num_procs: 56
}

HostObservation {
    time: Thu Jan 06 13:33:04 EST 2005
    type: PROC.CPU
    ip: 10.0.0.24
    name: sftp-server
    pid: 30039
    state: S
    parent_name: sshd
    ppid: 30038
}
```

The observation's *type* field is a generic descriptor for the type of anomaly detected at the host. Observations are divided into two semantic categories: continuous and discrete. Continuous observations correspond to slope anomalies, whereas discrete observations correspond to singular, categorical sensed events. Observations are subdivided into process-level (*PROC.**) and system-level (*SYS.**). The observation type hierarchy shown in Table 1 illustrates the type and granularity of information that is fed into TRAFEN.

3.3 Models

Leveraging the extensibility of the TRAFEN system, we have developed a suite of novel process models to perform event correlation, scoring and problem determination. TRAFEN models encapsulate all of the necessary logic for evaluating hypothetical tracks of observations from monitored hosts and network sensors. Since the TRAFEN infrastructure abstracts away the collection and mapping of host observations to process models, the administrator can concentrate on developing and experimenting with a set of underlying models by applying various state estimation techniques. For the purpose of autonomic host monitoring, each model is designed to track one of the following:

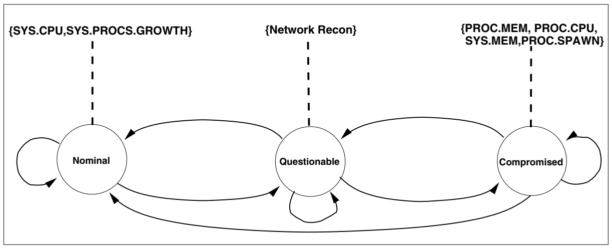

{SYS.CPU,SYS.PROCS.GROWTH}
{Network Recon}
{PROC.MEM, PROC.CPU, SYS.MEM,PROC.SPAWN}
Nominal
Questionable
Compromised

Figure 4. An example state-machine process model, $\mathcal{M}_3$, for detecting generic attack-related server compromises through host and network activity correlation.

- Isolated host runtime performance.
- Correlated network and host activity.
- Aggregate network status.

The TRAFEN engine evaluates incoming observations one at a time, and checks if they are related by some model-specific measure. Correlated events (events that are likely related) are gathered as tracks, which describe one possible host scenario. We have implemented several prototype models that embody the problem behavior of monitored entities with varying degrees of complexity and granularity. These initial models serve to reduce the overall rate of false positives in server failure detection.

We are currently working with process models that cover the following general scenarios:

1. *Network intrusion and host-level manifestations:*

   Models in this category analyze changes in host behavior after a network-level event relevant to that host (IP) has occurred. The underlying rationale for this design pattern is that host sensors will often report "exceptional events" that aren't actually alarming. Similarly, network-based sensors frequently generate false positives in response to innocuous network activity. Hence, there is often detrimental noise at both host and network levels. By tracking certain properties of reported host-level events, however, models can interpret finer-grained host activity and temporally correlate it

Table 1. User-space sensor observation types

Continuous types

PROC.MEM	Process memory utilization
PROC.CPU	Process CPU utilization
SYS.DISK.IO	Disk read/write operations
SYS.NET.RX	Incoming network traffic
SYS.NET.TX	Outgoing network traffic
SYS.MEM	System memory utilization
SYS.CPU	System CPU utilization

Discrete types

PROC.SPAWN	Spawning of a monitored process type (e.g., sh, httpd)
PROC.TERM	Termination of a monitored process type (e.g., sh, httpd)
SYS.PROCS.GROWTH	Growth in number of processes
SYS.NET.SOCKS.GROWTH	Growth in number of active network sockets

with pertinent network activity (by common IP). This should subsequently reduce the rate of false-positives and enable the system to arrive at more sensible hypotheses about the current state of a monitored host.

These models operate on two independent observation spaces, H and N, for the observable host and network events, respectively. The observations from each of these spaces at time t are denoted as $h_{1:t} = (h_1, h_2, \ldots, h_t)$ and $n_{1:t} = (n_1, n_2, \ldots, n_t)$.

For each observation, arriving at time t, the model computes $p(s_t | h_{1:t}, n_{1:t})$, the conditional probability of how likely the current event sequence of host and network observations indicates a modeled attack or failure scenario, s_t, for a monitored host. At each time, t, the TRAFEN engine feeds many different subsets of the observation sequence (tracks) to each model for evaluation. Over time, many tracks accumulate and are scored according to the probability $p(s_t | h_{1:t}, n_{1:t})$. Tracks with lower scores are eventually pruned, and those remaining continue on to the subsequent iteration, where they may grow.

Figure 4 shows a simple state machine process model, $\mathcal{M}_3$, with three generic server states: $Q_3 = \{Nominal, Questionable, Compromised\}$. It treats variants such as short-term system CPU spikes (*SYS.CPU*) and aggregate process count growth (*SYS.PROCS.GROWTH*) as indicators of steady-state server behavior. When one or more NIDS observations with the same destination IP as the server arrives, it transitions to the *Questionable* state, where it waits for a specified timeout. If this is followed by the spawning of a monitored process type, a jump in system memory, or a series of *PROC.CPU* or *PROC.MEM* observations, the *Compromised* state is entered. In this state, the probability of compromise, $p(s_t | h_{1:t}, n_{1:t})$, grows for each qualified host observation reported within the timeout.

2. *Isolated host failure or performance degradation:*

   Models in this category perform local diagnoses of host failures caused by system or service misconfigurations and application flaws. Such models do not make use of global network context, but the TRAFEN engine is still useful in making runtime assessments, as it can provide administrators with a concise and contextual view of the status of their servers.

We can configure TRAFEN to simultaneously evaluate observation streams using multiple models to formulate causal alternatives for each detected host failure or performance degradation. For instance, we could submit the host-network activity correlator model, $\mathcal{M}_3$, along with an isolated host degradation model, $\mathcal{M}_4$, to a TRAFEN instance. For simplicity, define $\mathcal{M}_4$ as a state machine $S_4 = (Q_4, \Sigma_4, \delta_4)$, with equivalent states as $\mathcal{M}_3$, i.e. $Q_3 \equiv Q_4$, and $\Sigma_4 = \{h_a, h_b, h_c, h_d, h_e, h_f, h_g\} \subseteq H$. Let $\Sigma_3 = \{n_a, n_b, h_a, h_b, h_c, h_d, h_e, h_f\}$, where $\{n_a, n_b\} \subseteq N$. Now suppose that TRAFEN receives the following sequence of host and network-level observations for an observed host (i.e., all share the same destination IP):

$$e_{1:12} = h_e h_f h_g h_a h_g n_a h_g h_g h_g h_g h_g n_b$$

Based on the scores generated by the two models over time, the following tracks would be returned as the combinations that most likely indicate the occurrence of their corresponding failure scenarios (s_3, s_4) on the observed host:

$$\begin{aligned} \mathcal{M}_3 \rightarrow &\quad h_e h_f h_a h_a n_a n_b \\ \mathcal{M}_4 \rightarrow &\quad h_e h_f h_g h_a h_g h_g h_g h_g h_g h_g \end{aligned}$$

The two models treat different event subsequences as indicators of s_3 and s_4, and grow their track likelihoods independently. Therefore, each model forms its own conclusions about the observation sequence. In this example, $\mathcal{M}_3$ generates the likelihood $p(s_3|h_{1:t}, n_{1:t}) = 0.55$ and $\mathcal{M}_4$ the likelihood $p(s_4|h_{1:t}) = 0.9$. TRAFEN subsequently concludes that this event sequence's target host is certainly "questionable", but that it is most likely facing some sort of an isolated failure, perhaps a misconfiguration. The observed network events (n_a, n_b), are most likely not associated with an attack.

The resultant multi-model likelihoods can be compared using a threshold, as in [8], to identify the failure and facilitate an autonomic response:

$$B < \frac{p(s_3|h_{1:t}, n_{1:t})}{p(s_4|h_{1:t})} < A$$

If the ratio falls below the threshold, B, then s_4 is deemed the fault scenario occurring in the observed environment. Alternatively, if it is greater than the threshold, A, then s_3 is deemed the fault scenario. If the ratio falls between A and B, however, the system waits until more observations are received. By adjusting A and B, we can control the sensitivity of the response mechanism and enforce a minimum confidence before taking any action.

4 Results

We have performed initial experiments of the system on a test-bed production network of heterogenous servers and workstations. Initial results have shown promise in our approach for detecting and qualifying server failures caused by network intrusions. In this section, we present the results from an experiment involving a prototype host-network activity correlation model, $\mathcal{M}_3$, described in 3.3.

4.1 Experimental Setup

The experimental testbed is a notional DMZ that consists of four Pentium III and IV servers, each running RedHat Linux and Apache HTTP server 2.0.40 or 2.0.52. A fifth machine on a separate network acts as an external attack host, from which a manual denial of service (DoS) attack is launched. Each server is instrumented with a user-space sensor, which runs continuously. In addition to aggregate system state, the sensors are configured to monitor the *httpd, bash,* and *sh* processes. The sensors' state extractor modules poll all of the monitored processes and system metrics once every four seconds. The sensor's evaluator modules smooth these continuous metric streams over a window of five samples.

The TRAFEN engine runs on a local Sun Fire 210 server with dual 1GHz UltraSPARC III processors and Solaris 9, and listens for observations from the monitored hosts over a dedicated TCP socket. TRAFEN also listens for observations from a mainstream network intrusion detection sensor (*Snort*), which monitors network traffic on the DMZ. Hence, we have two independent observation spaces.

The TRAFEN engine reports its real-time conclusions to a web-based user interface, which displays tracks of observations and associated scores. The scores (track likelihoods) are a quantitative measure between 0.0 and 1.0 of how likely a track of observations indicates an attack or failure scenario described by a submitted process model. Therefore, track scores are indirect, real-time measures of individual host condition with respect to criteria defined in the associated model. Each track score is also displayed qualitatively as a color-coded progress bar, which transitions from green to yellow to red based on model-specific score thresholds. Red typically represents a host that has most likely been compromised, and requires immediate attention from an administrator or an autonomic healing mechanism. By submitting multiple process models, we can effectively monitor each host for multiple scenarios, as demonstrated in 3.3. For this experiment, however, we load only our prototype host-network correlation model, $\mathcal{M}_3$.

During experimentation, we record the evolution of the track scores and corresponding progress bars for each monitored host. The underlying objective is to gauge the system's ability to reduce false positives during a multi-stage attack by correlating observations from both spaces and computing subsequent track scores that reflect the relative state of each host.

The simulated attack exploits a vulnerability in Apache webserver versions 2.0.40 through 2.0.52, which allows attackers to instigate a denial of service by flooding the server with specially-crafted HTTP GET requests [3]. This causes significant CPU and memory consumption, and can hang the entire server. This results in a DoS, since the Apache server never kills the connections while being attacked. Hence, the attack could be thwarted with minimal disruption if the parent server process (*httpd*) were killed in time and a backup web server brought online. The simulated attack sequence is as follows:

1. Using *nmap*, attacker initiates an asymmetric stealth port scan of the four DMZ servers from the external attack host to identify potentially vulnerable targets;

2. Attacker waits for all scan results and notices that each server is listening on port 8000;

3. Attacker waits an additional 30 seconds and launches an Apache DoS attack against one server (*host 3*), listening on port 8000;

4. Attacker waits 30 seconds and terminates the attack.

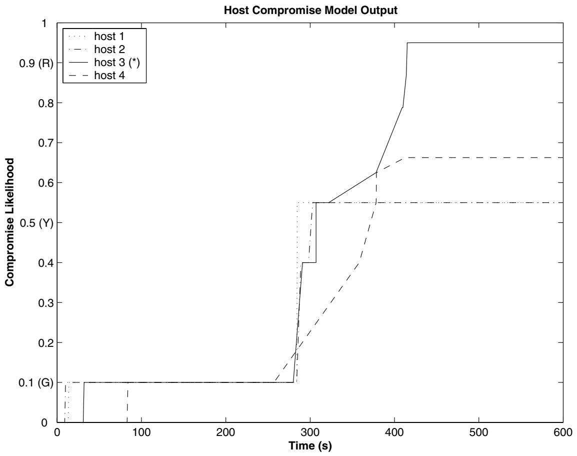


Figure 5. Progression of the estimated likelihood of compromise for four servers monitored by TRAFEN, using the prototype host-network model, $\mathcal{M}_3$. *host 3* **is designated as the victim server.**

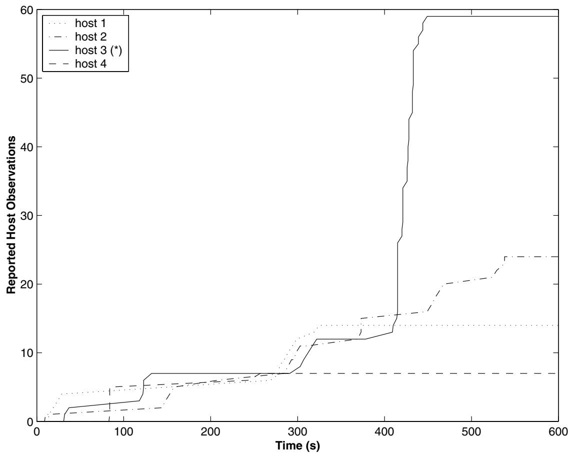


Figure 6. Host observations generated by each monitored host's user-space sensor and reported to TRAFEN.

4.2 Experimental Results

The sensors were run for a sufficient period of time prior to data collection and online analysis to allow for any startup transients to elapse. The time series in Figure 5 shows the evolution of the top track scores for each host over the duration of the experiment, approximately 10 minutes. The y-axis is the likelihood of compromise, as determined for each server by the submitted process model, $\mathcal{M}_3$. For this model, scores (likelihoods) below 0.5 are denoted as low (green) severity, between 0.5 and 0.9 as medium (yellow), and above 0.9 as high (red).

This graph shows the classification and detection performance of the model before, during, and after the simulated manual DoS attack. Initially, the state estimate for each server is *Nominal*. Soon after the scan initiation (280 seconds into experiment), each server transitions to the *Questionable* state. When the DoS flood is launched against *host 3* at 396 seconds, its score suddenly jumps again, quickly moving it to the *Compromised* state. The other three servers continue normal operation, and remain in the *Questionable* state. At 415 seconds, just 19 seconds after the start of the DoS flood, the victim's likelihood has grown to 0.95. At this stage, an administrator or autonomic healing mechanism could concentrate on mitigating the compromise on *host 3* by terminating the pertinent *httpd* process(es), which are clearly identified in the host observations, as shown in 3.2.

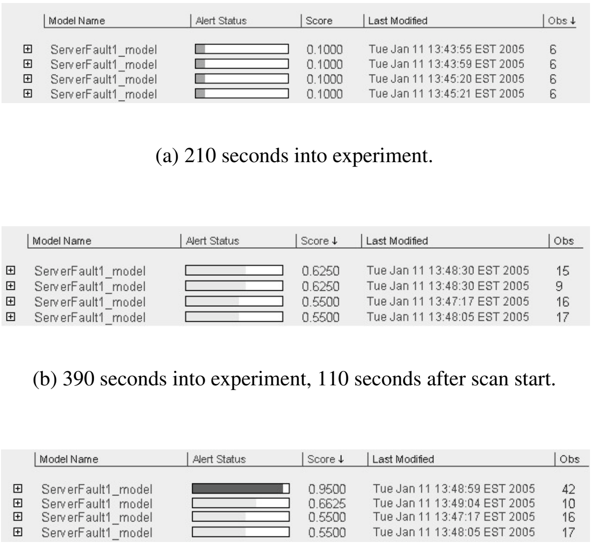
Figure 6 shows the aggregate growth in host observations reported to TRAFEN over the same time frame. This graph illustrates how the four servers generate similar levels of host observations until the actual DoS commences. Most apparent is the sudden and significant jump in aggregate host observations generated by the victim server (*host 3*) shortly after attack initiation. Examination of this trend and the graph in Figure 5 reveals the clear correlation between simple, short-term anomalies in host behavior and suspicious network reconnaissance activities. In this trial, the user-space sensor provides the underlying indicators that drive the victim server's state estimate to *Compromised*, enabling rapid detection while the attack is still underway.

Table 2 provides a drill-down comparison of host-level behavior during the experiment, as reported by each server's user-space sensor and the deployed *Snort* sensor. There are several data points within this table that plainly differentiate the victim, *host 3*, from the unaffected servers. Most profound is the exclusive prevalence of *PROC.MEM* observations, which correspond to the acute growth in memory utilization of the overloaded Apache server (*httpd*) processes in response to the HTTP request flood. Likewise, the *PROC.SPAWN* observations correspond primarily to the forking of child server processes for handling the incoming requests. It is especially worth noting how the distribution of *Snort* reconnaissance observations is relatively uniform across the four servers. Therefore, despite each server being equally vulnerable to the same exploit, and experiencing comparable attack precursors, only the victim host produces traces of deviant service behavior, subsequently elevating it to the *Compromised* state.

Figure 7 further demonstrates the system's ability to prioritize manual or autonomic responses to the attack. These are screen shots of the front-end TRAFEN display, a real-

Table 2. Distribution of all host and network observation types reported to TRAFEN

Observation	host 1	host 2	host 3(*)	host 4
PROC.MEM	0	0	22	0
PROC.CPU	0	1	1	0
SYS.MEM	0	0	7	0
SYS.CPU	12	20	11	2
PROC.SPAWN	0	1	13	2
SYS.PROCS.GROWTH	1	2	4	3
Snort	3	2	3	4



	Model Name	Alert Status	Score	Last Modified	Obs ↓
⊞	ServerFault1_model		0.1000	Tue Jan 11 13:43:55 EST 2005	6
⊞	ServerFault1_model		0.1000	Tue Jan 11 13:43:59 EST 2005	6
⊞	ServerFault1_model		0.1000	Tue Jan 11 13:45:20 EST 2005	6
⊞	ServerFault1_model		0.1000	Tue Jan 11 13:45:21 EST 2005	6

(a) 210 seconds into experiment.

	Model Name	Alert Status	Score ↓	Last Modified	Obs
⊞	ServerFault1_model		0.6250	Tue Jan 11 13:48:30 EST 2005	15
⊞	ServerFault1_model		0.6250	Tue Jan 11 13:48:30 EST 2005	9
⊞	ServerFault1_model		0.5500	Tue Jan 11 13:47:17 EST 2005	16
⊞	ServerFault1_model		0.5500	Tue Jan 11 13:48:05 EST 2005	17

(b) 390 seconds into experiment, 110 seconds after scan start.

	Model Name	Alert Status	Score ↓	Last Modified	Obs
⊞	ServerFault1_model		0.9500	Tue Jan 11 13:48:59 EST 2005	42
⊞	ServerFault1_model		0.6625	Tue Jan 11 13:49:04 EST 2005	10
⊞	ServerFault1_model		0.5500	Tue Jan 11 13:47:17 EST 2005	16
⊞	ServerFault1_model		0.5500	Tue Jan 11 13:48:05 EST 2005	17

(c) 415 seconds into experiment, 19 seconds after attack start.

Figure 7. Front-end displayed host tracks and severities.

time dashboard for the track scores and severity levels of each monitored host. The three screen shots were acquired over the course of the experiment and convey to the end user the same information as in Figure 5.

Overall, the results from this initial experiment demonstrate how our non-intrusive user-space host sensors can, with fairly minimal effort, be tuned to adequately detect and report low-level abnormalities that are essential to high-level intrusion detection.

5 Analysis and Future Work

We are presently working on the implementation and testing of the existing models and sensors. A much larger-scale experiment is planned, which will entail a comprehensive evaluation of the system's scalability (as defined in 2.3) when deployed across dozens of virtual, heterogenous servers. We are confident, however, that these initial results from a network DMZ indicate the potential scalability improvement offered by our system.

We are experimenting with a more extensive range of process models that describe system behavior at different levels of abstraction. The objective is to determine which types of models perform best with minimal training or calibration to enable rapid deployment across multiple networks. This is also the case for the user-space sensor, which we will continue to build on by adding additional monitored performance metrics.

Additional work in progress:

- Integrate a feedback loop to administer self-healing actions based on the broader enterprise awareness attained through the existing system. This autonomic healing mechanism will craft suitable responses to protect mission-critical services based on the conclusions generated by the process models. This will comprise a response module integrated into the TRAFEN platform, and a host-based effector. The effectors will execute TRAFEN response directives, such as the termination of compromised processes.

- Implement a level-2 aggregator model. Part of the autonomic utility of this system is to dynamically fine-tune the anomaly detection parameters, such as metric thresholds, in order to enhance runtime performance and reduce the occurrence of possible false positives at higher levels in the system. This process model will monitor entire groups of servers and track runtime statistics of host observation frequencies to derive a global baseline. This will improve the detection accuracy of individual system behavior models by helping identify outliers amongst larger server populations.

6 Conclusions

In this paper, we have presented a promising new methodology and supporting architecture to automate rapid problem determination for multiple servers. By combining the scalability, robust data collection, model abstraction, and hypothesis management capabilities of PQS, we were able to apply a fundamentally less complex and intrusive host sensor paradigm to rapidly detect a multi-stage denial of service attack against a server network.

Our system reduces false positives by using process models to probabilistically correlate host-level symptoms of performance degradation with network context. This was demonstrated in an experiment involving a host and network activity correlator model. Through temporal and spatial correlation, the submitted model determined with high confidence that the attacked server had been compromised, disambiguating it from the unaffected servers.

Our approach shows promise in its ability to adequately reduce false positives in a heterogenous server network. It is an important step forward towards more automated server and service awareness in large networks. However, more work remains to be done to fully assess the scalability of the system's runtime and detection performance in much larger environments.

References

[1] V. Berk, W. Chung, V. Crespi, G. Cybenko, R. Gray, D. Hernando, G. Jiang, H. Li, and Y. Sheng. Process Query Systems for surveillance and awareness. In *Proceedings of the Systemic, Cybernetics and Informatics (SCI2003) conference, Orlando Florida*, July 2003.

[2] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. Remote repair of operating system state using Backdoors. In *Proceedings of the International Conference on Autonomic Computing*, May 2004.

[3] Common Vulnerabilities and Exposures List. CVE advisory CAN-2004-0942 apache webserver denial of service vulnerability. Technical report, The MITRE Corporation, 2004.

[4] G. Cybenko, V. H. Berk, V. Crespi, R. S. Gray, and G. Jiang. An overview of process query systems. In *Proceedings of SPIE Vol. 5403 Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense III , Orlando, Florida*, April 2004.

[5] G. Dudley, N. Joshi, D. M. Ogle, B. Subramanian, and B. Topol. Autonomic self-healing systems in a cross-product IT environment. In *Proceedings of the International Conference on Autonomic Computing*, May 2004.

[6] T. Ferreto, C. D. Rose, and L. D. Rose. RVision: An open and high configurable tool for cluster monitoring. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[7] S. Forrest, S. Hoffmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[8] G. Jiang and G. Cybenko. Temporal and spatial distributed event correlation for network security. In *Proceedings of the American Control Conference (ACC), Boston, MA*, July 2004.

[9] O. P. Kreidl and T. M. Frazier. Feedback control applied to survivability: A host-based autonomic defense system. *IEEE Transactions on Reliability*, 53(1):148–166, March 2004.

[10] J. Lerner. *Advanced System and Security Monitoring - Achieving Complete System Control*. JJ Labs Inc., 2003. White Paper, http://www.jjlabs.com.

[11] NimSoft. *NimBUS for Server Monitoring*. Solution Overview Paper, http://www.nimsoft.com.

[12] S. Parker. A simple equation: IT on = business on. *The IT Journal, Hewlett Packard*, 2001.

[13] M. J. Sottile and R. G. Minnich. Supermon: A high-speed cluster monitoring system. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[14] K. Theriault, W. Farrel, H. Henri, D. Kong, and W. Nelson. The SARA experiment: Coordinated autonomic defense against an e-mail-borne virus. In *Proceedings of the 2002 IEEE Workshop on Information Assurance*, June 2002.