Community Epidemic Detection using Time-Correlated Anomalies

Adam J. Oliner, Ashutosh V. Kulkarni, and Alex Aiken

Stanford University*
{oliner, ashutosh.kulkarni, aiken}@cs.stanford.edu

Abstract. An *epidemic* is malicious code running on a subset of a *community*, a homogeneous set of instances of an application. Syzygy is an epidemic detection framework that looks for time-correlated *anomalies*, i.e., divergence from a model of dynamic behavior. We show mathematically and experimentally that, by leveraging the statistical properties of a large community, Syzygy is able to detect epidemics even under adverse conditions, such as when an exploit employs both mimicry and polymorphism. This work provides a mathematical basis for Syzygy, describes our particular implementation, and tests the approach with a variety of exploits and on commodity server and desktop applications to demonstrate its effectiveness.

Keywords: epidemic detection, anomalies, community

1 Introduction

Consider a set of instances of an application, which we call a *community*. Two examples of communities are all the mail servers in an organization or all the browsers on a cluster of workstations. Assume some subset of these instances, or *clients*, are compromised and are running malicious code. The initial breach (or breaches) went undetected and the existence of the exploit is unknown, so the malicious code may continue running indefinitely, perhaps quietly stealing computing resources (as in a zombie network), spoofing content, denying service, etc. We present a method for detecting such situations by using properties of the aggregate behavior of the community to reliably identify when a subset of the community is not behaving properly.

A client is either *healthy* and exhibits correct behavior or *infected* and exhibits incorrect behavior; our method detects *epidemics*, meaning when a subset of the community is infected. The user specifies what constitutes correct operation for individual clients by providing a *model*, which may be incomplete (omit correct behaviors), or unsound (admit incorrect behaviors), or both. For example, a community of web servers may be modeled by the typical distribution of response times each provides. The class of attacks we want to detect are those that cause undesirable deviation from normal behavior, regardless of the attack vector (e.g.,

* This work was supported in part by NSF grants CCF-0915766 and CNS-050955, and by the DOE High-Performance Computer Science Fellowship.

buffer overrun, insider attack, or hardware tampering). Our focus is on detecting epidemics in a community composed of instances of a specific application, rather than the entire system or individual clients in the community, and this distinction leads to a different approach.

We describe an implementation of an epidemic detector, called Syzygy, that applies two main insights: (i) even if a single noisy model cannot reliably judge the health of a client, we can reduce the noise by averaging the judgements of many independent models and (ii) epidemics exhibit time-correlated behavior that is impossible to detect on a single client. Our method effectively leverages the statistical properties of a large community to turn noisy models into reliable community detectors and uses the temporal properties of an epidemic as a means for better detecting it.

Syzygy monitors each client's behavior and reports *anomaly scores*, which quantify the divergence of recent behavior from the model. For example, a client whose recent response times are unusually high may report a score that is above average (anomalous). Syzygy then computes the numerical average of all clients' scores and checks whether this *community score* exceeds a threshold. By doing these computations properly (see Section 3), we can make strong theoretical guarantees about our ability to overcome model noise and detect epidemics. Intuitively, we expect anomalies on individual clients in a large community to be common, but we do not expect anomaly scores from multiple clients to be strongly correlated in time, absent an epidemic.

We describe and analyze Syzygy's detection algorithm mathematically in Section 3. In our evaluation, we focus on the following questions:
—*Can Syzygy detect epidemics under realistic conditions?* In Section 4, we demonstrate that our method can leverage the community to detect a variety of epidemics in a cluster of commodity web servers even given noisy, incomplete client models. Syzygy does not require source code or specially compiled binaries.
—*How do client and community characteristics affect performance (i.e., false positives)?* In Section 5, we deploy Syzygy on the web browsers of a campus network and show that, despite very different client systems and user behaviors, healthy community behavior is a stable, reliable signal that is unlikely to generate excessive false positives (our deployments generated none). Indeed, as the community grows, Syzygy approaches a 100% detection rate with no false positives; given a sufficiently large training set and community, one can specify an acceptable false positive rate *a priori* and with high confidence. Even communities of only a dozen clients exhibit desirable properties. See Sections 3.3, 4.2, and 5.2–5.3.
—*What kinds of epidemics can Syzygy detect?* In Section 6, we conduct simulation experiments using commercial, off-the-shelf software and artificially powerful exploits (e.g., capable of nearly perfect mimicry) and demonstrate that the community enables Syzygy to detect epidemics under a variety of adverse conditions. Exploits may change their source code, perform different malicious actions, or even use a different vector of infection across clients (see Section 3.2).
—*How good must client models be and how easy is it to acquire such models?*

Syzygy works on top of existing client-based anomaly detectors, dampening noise and providing sensitivity to time-correlated behavior. Syzygy requires only that anomaly scores are mostly independent across healthy clients and higher, on average, for infected clients; the method is agnostic to what measurements are used to construct these scores.

Throughout the paper—using math, deployments, and simulations—we show that, in a large community, even simple, noisy models are sufficient for reliable epidemic detection. We conclude with a discussion of the issues involved with building a larger-scale deployment (Section 7). Many real security infrastructures are a constellation of tools; working in concert with other detection and response tools, and with low overhead and few practical requirements, Syzygy provides both new and more reliable information about epidemics.

2 Related Work

Syzygy detects malicious software running on clients in a community (epidemics) even under typical real-world constraints: the client model is incomplete, information about communication (network activity) is unavailable, and measurements are noisy. It may be impossible, given social engineering and insider attacks, to prevent all security breaches; a strength of Syzygy is that it can detect the bad behavior that follows a breach. In situations where the total damage is integral over time and the size of the infected community—such as when an exploit is stealing resources—the ability to detect such epidemics is crucial.

Anomaly-based intrusion detection has a long history [5, 27, 28, 29, 31, 35]. A commonly held view is that anomaly detection is fundamentally limited by the mediocre quality of the models that can be obtained in practice and therefore must necessarily generate excessive false positives in realistic settings (see, e.g., [2]). We agree with the gist of this argument for single clients, but we show in this paper that an appropriate use of a community can make strong guarantees even with noisy models.

Crucial, however, is how the community is used. Most previous systems that use a community at all use it only to correlate alarms generated locally on each client—the difficulty is that the alarm/no alarm decision is still made on the basis of a single client. Alert-correlation systems then try to suppress the resulting false alarms by correlating alarms from other clients or different detectors [4, 13, 36]. Other collaborative detection efforts that raise alarms only on individual clients include heterogeneous network overlays [44] and network anomaly detectors, such as by using cumulative triggers [15, 16] or alarm aggregation and correlation [1, 17, 32, 41]. Some work also uses correlation to characterize attack scenarios and causal flow [19, 26, 34].

Syzygy is fundamentally different from all of these systems in that it uses the aggregate behavior of the community to decide whether to raise an alarm for the community, not individual clients. The ability to make alert decisions based on analyzing the combined behavior of multiple clients is what gives Syzygy strong theoretical and practical properties that are absent from all previous work. There is prior work for file systems [43] and peer-to-peer networks [22, 23] that

generate alerts based on aggregate behavior, but these do so without utilizing the statistical benefits of a large community.

Another category of work uses the community simply to gather data more quickly or to spread the burden of monitoring among many clients. For example, the Application Communities project [21] uses the community to distribute work; everything could be done on a single client, given more time. Syzygy uses the community in both these ways, as well; in contrast, however, it also looks for time-correlated deviations from normal behavior, which is not possible on a single client.

Syzygy was originally a detection component of the VERNIER security architecture [20]. Syzygy's role is to monitor instances of a target application for signs of infection: attacks on the security infrastructure or other applications within the client system, problem diagnosis, and reaction to the intrusion are all the responsibility of other VERNIER components. Among the various VERNIER detectors, Syzygy is specifically looking for time-correlated activity, as might be expected from a propagating worm or a coordinated attack. This specialization allows Syzygy to be small, lightweight, and asymptotically ideal while using the community in a novel way.

There are also uses of the community for tasks other than detection, such as diagnosing problems by discovering root causes [39] and preventing known exploits (e.g., sharing antibodies) [2, 3, 25]. Although other parts of VERNIER employ such measures, our focus is on detection.

3 Syzygy

Consider a community of n clients in which we wish to detect epidemics. During training, Syzygy observes the normal operation of the clients and builds a *model* (see Section 3.1). It is important to note that the specific choice of model is independent from the rest of Syzygy's operation; the only requirement is that the model produces an *anomaly signal* according to the constraints in Section 3.2.

While subsequently in monitoring mode, Syzygy periodically collects the most recent value of the anomaly signal (the *anomaly score*) from each client and checks whether the community's average anomaly score exceeds a threshold V. If so, Syzygy reports an *epidemic*. The properties of the anomaly signal are such that, given a large community, Syzygy can compute the threshold automatically at runtime and is insensitive to minor variations in this parameter. We explain these properties mathematically in Section 3.3 and support them experimentally in Sections 5.2 and 6.3.

3.1 Model

When applying our method to detect epidemics in a community, the user selects an appropriate client *model*, which uses some combination of signals that can be measured on individual clients to quantify how surprising (anomalous) recent behavior is. We require only that the model generate anomaly scores that are mostly independent across healthy clients and that it quantify how surprising recent behavior is, compared with historical behavior or a theoretical baseline.

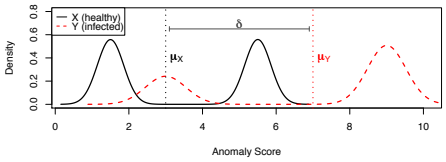

Fig. 1. An illustration of anomaly signals. Neither X nor Y are normally distributed, but $\mu_Y > \mu_X$, as required. The exploit may sometimes look "normal".

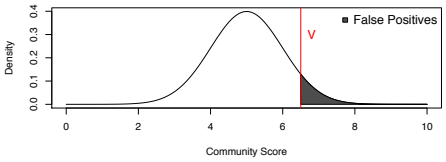

Fig. 2. A distribution of healthy community scores using hypothetical data. The threshold V determines what fraction of scores result in false positives.

The model for a community of servers might characterize normal behavior according to performance (see an example using request response times in Section 4), while the model for a community of web browsers might use code execution paths (see examples using system calls in Sections 5 and 6). The example models used in this paper could easily be refined or replaced with alternatives to match the attacks we want to detect: call stack content [8], execution traces [10], call arguments [24], remote procedure calls [12], etc.

3.2 Anomaly Signal

The anomaly signal decouples the choice of model from the rest of the system; any model that satisfies the properties explained in this section may be used with Syzygy. Each client keeps the server apprised of the client's *anomaly score*, the current value of the client's *anomaly signal*. This score is a measure of how unusual recent behavior is compared to a model of client behavior: a higher score indicates more surprising behavior than a lower score. (This is sometimes called the IS statistic [18] or behavioral distance [11].)

The distribution of anomaly scores generated by a healthy client (X) must have a mean (μ_X) that is less than the mean (μ_Y) of the anomaly score distribution of an infected client (Y), so we require $\mu_Y > \mu_X + \delta$. The larger the δ, the better, though any positive δ will suffice. Figure 1 illustrates two valid anomaly signal distributions, where X and Y are random variables such that both have finite mean and finite, positive variance.

More generally, let the anomaly scores from healthy client i, denoted a_i, be distributed like X_i (written $a_i \sim X_i$) and let $a_i \sim Y_i$ when client i is infected. Assume, without loss of generality, that all clients have the same distribution, i.e., let $X_i \sim X$ and $Y_i \sim Y$. The distributions may be standardized to enforce this assumption, because only the mean and variance are relevant to our asymptotic results. If infected behavior does not differ from normal behavior, then δ will be unacceptably small (even negative); this can be resolved by refining the model to include more relevant signals or adjusting the model to amplify surprising behaviors. In this paper, we use two simple models (see Sections 4.1 and 5.1) that share a similar anomaly score computation (see Section 4.1), and both provided sufficiently large δ values to detect a variety of exploits.

3.3 Epidemic Detection

The Syzygy server computes the average anomaly score among the active clients; this *community score* C represents the state of the community. If $C > V$, for a tunable threshold V, the server reports an epidemic. Consider a healthy community of n clients and let $a_i \sim X$. Then, by the Central Limit Theorem, as $n \to \infty$, the community scores are distributed normally with mean μ_X and variance $\frac{\sigma_X^2}{n}$:

$$C = \text{average}_i(a_i) = \frac{1}{n}\sum_i (X) \sim \text{Norm}(\mu_X, \frac{\sigma_X^2}{n}).$$

When $E(|X|^3) = \rho < \infty$, where $E()$ denotes expected value, convergence happens at a rate on the order of $\frac{1}{\sqrt{n}}$ (Berry-Esséen theorem). Concretely, let $C' = C - \mu_X$, and let F_n be the *cumulative distribution function* (cdf) of $\frac{C'\sqrt{n}}{\sigma_X}$ and Φ the standard normal cdf. Then there exists a constant $B > 0$ such that $\forall x, n$, $|F_n(x) - \Phi(x)| \leq \frac{B\rho}{\sigma_X^3\sqrt{n}}$.

Consider now when some number of clients $d \leq n$ of the community have been exploited. The community score, as $n, d \to \infty$, will be

$$C = \frac{1}{n}\left(\sum_{i=1}^{n-d} X + \sum_{i=1}^{d} Y\right) \sim \text{Norm}\left(\frac{(n-d)\mu_X + d\mu_Y}{n}, \frac{(n-d)\sigma_X^2 + d\sigma_Y^2}{n^2}\right).$$

The rate of convergence guarantees that we get this asymptotic behavior at relatively small values of n and d, and even when $d << n$; in Section 6 we support this fact experimentally.

The threshold V must be set given the community size (n) and given the mean (μ_X) and standard deviation (σ_X) of the healthy client anomaly scores, but without knowing the size (d) and distribution (μ_Y and σ_Y) of the infected population, because those are unknown at runtime. We can pick any positive V between σ_X^2/n and $(\sigma_X^2/n) + \delta$ and guarantee that there exist n and d that give an arbitrarily high probability of perfect detection (FP=FN=0). Without knowing δ, however, the best strategy is to pick the lowest value of V such that the false positive rate is acceptable. Using the following analysis, we can compute and adjust V at runtime based on known quantities and a specified false positive rate; we do this using data from real deployments in Sections 4.2 and 5.2.

The expected rate of false positives is the fraction of the community scores in a community with no infected clients that falls above V. (See Figure 2.) This is precisely the value of the parametrized Q-function, the complement of the normal cdf: $Q(\alpha) \equiv \frac{1}{\sqrt{2\pi}}\int_\alpha^\infty e^{-\frac{x^2}{2}}dx$. Let $H \sim \text{Norm}\left(\mu_X, \frac{\sigma_X^2}{n}\right)$ be the distribution of community scores in a healthy community of size n. The probability that a randomly selected community score will be a false positive is $\text{FP} = P(C > V) = Q\left(\frac{(V-\mu_H)\sqrt{n}}{\sigma_H}\right)$. Table 1 lists the significant terms and metrics used in this paper.

This analysis relies on two modest assumptions. First, the parameters μ_X and σ_X must characterize the future distribution of anomaly scores. A model

Term	Meaning
n	The total number of active clients in the community.
d	The number of infected clients in the community.
W_i	The size of the recent window on client i. We use $W_i = 1000$ measurements.
T_i	The silence threshold on client i. If the application records no measurements for T_i seconds, Syzygy generates a hiaton; if a client reports no anomaly scores for $2T_i$ seconds, the server marks it inactive.
a_i	Anomaly score. The instantaneous value of the anomaly signal $A_i(t)$ on client i.
X, Y	The distributions of anomaly scores for healthy (X) and infected (Y) clients.
C	Community score: average of the most recent anomaly scores from active clients.
V	The epidemic threshold. If $C > V$, Syzygy reports an epidemic.
δ	Defined as $\mu_Y - \mu_X$. Intuitively, the average distance between anomaly scores generated by healthy versus infected clients. One kind of mimicry attack drives δ toward zero.
r	The rate of a rate-limited mimicry attack: the application appears healthy a fraction $1 - r$ of the time and infected a fraction r of the time.
TP	True positive rate or detection rate. $P(E\|\neg H)$.
TN	True negative rate. $P(\neg E\|H)$.
FP	False positive rate, or Type I classification error rate. $P(E\|H)$.
FN	False negative rate, or Type II classification error rate. $P(\neg E\|\neg H)$.
F1 Measure	A summary metric with precision and recall weighted equally: $\frac{2TP}{2TP+FP+FN}$.

Table 1. A reference table of the terminology used in this paper. Let E be the event that Syzygy reports an epidemic and let H be the event that the community is healthy.

that is out-of-date or produced with biased training data, for example, may produce anomaly scores inconsistent with the expected distribution. In Section 6.4 we explore the impact of using on one system a model produced for a different one and in Section 5.2 we show that even relatively heterogeneous machines produce predictable community score distributions. It is straightforward to detect when observed behavior disagrees with expectation, and the solution is to retrain the model. Second, during normal operation, client anomaly scores should be mostly independent. In situations like a network-distributed software upgrade, innocuous dependencies may cause correlated behavior (i.e., correlated behavior without a malicious cause, which is our definition of a false positive). Indeed, it is indistinguishable from an attack except that one change to the software is authorized and the other is not. Such false alarms are easily avoided by making information about authorized changes to monitored applications available to Syzygy. Other sources of accidentally correlated behavior are quite rare; we observed no false alarms at all in a deployment with real users (see Section 5).

4 Detection Experiments

We first test Syzygy's ability to detect epidemics in a community using a cluster of 22 machines running unmodified instances of the Apache web server. Each machine has four cores (two dual core AMD Opteron 265 processors), 7 GB of main memory, and the Fedora Core 6 distribution of Linux. Each client serves streams of requests generated by a workload script. The workload generator, at exponentially distributed random times, makes requests from a list of 178 available HTML and PHP pages that includes several pages that do not exist and two pages for which the requester does not have read permission. We run the workload generator for 100,000 requests (∼2.8 hours) to train the model, then use those same training traces to set V so that we expect to get one false positive per week (see Section 3.3 for how we do this; also see Section 5.2 for more on false positives). We use Apache's existing logging mechanisms to record measurements (e.g., response times).

For this community, we aim to detect the following classes of attack: denial of service (DoS), resource exhaustion, content spoofing, and privilege escalation. Thus, we pick a client model that is likely to detect such attacks (see Section 4.1). We test Syzygy with two DoS attacks that prevent Apache from serving 1% or 10% of requests, at random, respectively; two resource exhaustion attacks that allow Apache to continue serving requests but gradually consume memory or CPU time, respectively; three content spoofing attacks that cause (i) PHP pages to be served in place of previously non-existent pages, (ii) PHP pages to be served in the place of certain HTML pages, or (iii) HTML pages to be served in place of certain PHP pages; and a privilege escalation attack that makes all page accesses authorized (no 403 Errors). We find that Syzygy can achieve high detection rates for these attacks with no false positives (see Section 4.2).

The clients in these experiments are homogeneous; in Section 5, we explore the effects of heterogenous hardware and varying user behavior with a deployment using an interactive application (the Firefox web browser). Section 6 contains additional experiments, in a more controlled environment, that explore the properties of much larger communities (thousands of clients) and more advanced exploits (capable of various degrees of mimicry).

4.1 Model

Assume that our security goal for this community is to ensure that clients are serving requests according to expected performance; that is, the request response behavior should be consistent over time. During training, the model computes a frequency distribution of request response times and the maximum observed time between consecutive requests. This is just one choice of model and is not intrinsic to Syzygy.

When a request is made of the server, the model increments the counter associated with the response time s in a table indexed by response times (10 μsecond precision). From this frequency distribution, we compute a density function S_i by dividing each entry by the total number of observed response times. Thus, $S_i(s)$ is the fraction of times that response time s was observed on client i.

To incorporate timing in the model, which can help identify the absence of normal behavior (such as during a denial of service attack), we record the time between the start of each consecutive pair of requests. The model measures these times only when the application is *active*. A client is active when it reports its first anomaly score and becomes *inactive* after reporting an anomaly score accompanied by the `END` message. (See below for when this token is generated.) From these data, we set a *silence threshold* T_i for each client i, which we initially pick to be the maximum time between any two consecutive requests.

Monitoring On the client, Syzygy monitors all requests made to the application. In addition, Syzygy may inject two kinds of artificial measurements into the sequence. The first, called `END`, indicates that the application has terminated (switched to *inactive*); Syzygy generates an `END` token when the application exits cleanly, terminates abruptly such as due to an error, or when the Syzygy client is closed cleanly. If an active client stops reporting scores for longer than the *timeout threshold*, currently set to $2T_i$ seconds, then the Syzygy server marks that client inactive without fabricating a token. The second artificial measurement, a *hiaton* [37] denoted `X`, indicates that no measurements were generated for longer than T_i seconds, including any `X`s produced via this process. In other words, at the start of each request, a timer starts; when this timer exceeds T_i, Syzygy generates a hiaton and resets the timer.

Each client maintains a window of the most recent W_i request response times, including the fabricated hiatons and `END` tokens. From this window, we compute the density function R_i, analogous to S_i, above. Thus, $R_i(s)$ is the fraction of times measurement s appears in the previous W_i measurements on client i.

Anomaly Signal Let a_i be the most recent anomaly score and W_i be the size of the recent window for client i. The units of a_i and W_i may depend on the particular choice of model, but should be consistent across clients. In this paper, we measure the anomaly signal in bits and the window size in number of measurements. Our implementation computes a_i using Kullback-Liebler (KL) divergence with a base-2 logarithm. Roughly, this measures the information gained by seeing the recent window, having already observed the historical behavior. Specifically, over the measurements s in the density function for the recent window ($s \in R_i$), we have $a_i = \sum_s R_i(s) \log \frac{R_i(s)}{S_i(s)}$.

This computation can be updated incrementally in constant time as one measurement leaves the recent window and another enters it. To prevent division by zero, the measurements in the recent window are included in the distribution S_i. By default, each client reports this score whenever there is new information available to the model (e.g., a request or hiaton), but it is straightforward to add feedback or batching to the client-server protocol to curb communication traffic (we do so in Section 5.3).

4.2 Results

Figure 3 shows the results of our detection experiments; there were no false positives in these experiments and detection latency was never more than a

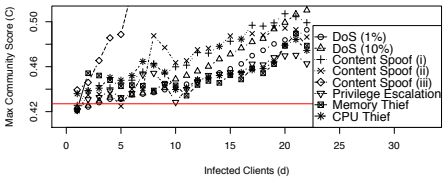

Fig. 3. Syzygy detected all of the attacks once the infection size was sufficiently large. The horizontal line is the epidemic threshold V.

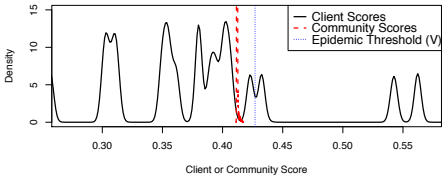

Fig. 4. Our client model is incomplete and noisy; anomalous behavior is common. The community scores, however, are extremely steady.

couple of seconds. Although some attacks are difficult to detect when only a few machines are infected (low d), Syzygy is able to correctly detect each attack once a sufficiently large number of clients are infected. In the case of the third (iii) content spoof attack, the behavior is anomalous enough on even a single client for our simple response time model to detect it; this is not true for most of the other attacks, meaning the community was crucial.

We achieved these high detection rates despite the fact that our behavior model was incomplete and noisy. Figure 4 shows part of the distribution of anomaly scores reported by individual healthy clients. In fact, these values ranged as high as 0.8 but we have truncated the graph for readability. In contrast, however, note that the healthy community scores stayed within a very small range (the dashed red line is actually a very slim Gaussian). The epidemic threshold V is the dotted line to the right of the cluster of community scores. Because the community scores are such a stable signal, they enable Syzygy both to reliably provide a low false positive rate and to be sensitive to minor—but not isolated—changes in client behavior.

In the subsequent sections, we discuss the benefits of distributed training, the effects of heterogenous hardware and user behavior, performance and overhead on a real network deployment, predicting and setting the false positive rate, performance in communities with thousands of clients, and Syzygy's robustness against tainted training data and advanced exploit behavior (like mimicry).

5 Deployment Experiments

For practical use, our method assumes that (i) a real deployment can scale to large numbers of clients across a realistic network topology and (ii) despite minor client variations, such as hardware and configuration differences, healthy anomaly score distributions are similar across clients. We verify that these assumptions hold in practice by deploying Syzygy on several dozen Linux workstations on a university campus. Most of these machines were 3.0 GHz Intel Core2 Duos with 2 GB RAM and the CentOS 5 operating system; exceptions include two laptops and (briefly) the Syzygy server, itself. Syzygy monitored the Firefox web browser via `strace` on Linux. Over the course of these two weeks of experiments, Syzygy reported no false positives.

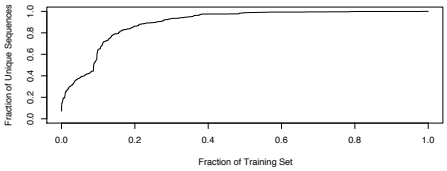

Fig. 5. Distributed training happens quickly: 25% of the data exhibits 90% of the unique sequences. Retraining a model (e.g., after a software upgrade) is efficient.

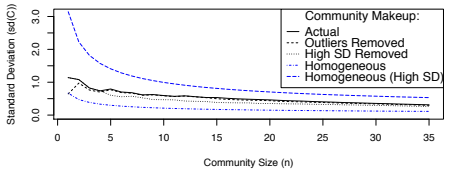

Fig. 6. Community scores converge in real data; variance comes from client variance, not system configuration or workload heterogeneity.

5.1 Model

In the next two sections, we use a model of client behavior (different from Section 4) that uses short sequences of a program's system calls. This information can be gathered with low overhead and has been shown to be useful [9, 14]. We use sequences of six system calls to be consistent with previous work [7, 14, 22], but instead of using one of the existing *stide* or *t-stide* algorithms [33], the model uses an information theoretic approach with several additional modifications. During training, Syzygy computes a frequency distribution of system call sequences of length six and the maximum observed time between consecutive system call invocations. The computations are extremely similar to Section 4.1, but use system call sequences as measurements, instead of request response times.

Whenever a system call is invoked, the model concatenates the name of the call onto a sequence consisting of the previous five and increments the counter associated with that sequence. For example, on Mac OS X, while executing the command `echo hi`, we generate the following period-delimited sequence:
$s =$ `sigaction.writev.read.select.select.exit`.
Even when idle, many applications will continue to invoke system calls (e.g., polling for new work or user input). This behavior acts as a kind of heartbeat for the program, and its absence indicates unusual behavior just as much as the presence of, say, unusual system call sequences. For example, during one such execution of `echo hi`, the maximum time between system call invocations, according to `dtrace`, was 375 μs.

Using this kind of information about call sequences and timing, we construct a model analogous to the one for request response times in Section 4.1. The only differences are that the tables used to construct S_i and R_i are indexed by sequences and the recent window W_i has units of sequences. The anomaly signal is computed as described in Section 4.1.

5.2 Distributed Training

Over a period of roughly two weeks, we collected normal usage traces from 35 active clients. During the day, a median of 8 clients were active at a time. The first week of these traces is our training data and contains more than 2.2 billion sequences, of which approximately 180,000 are unique. As shown in Figure 5, most of the sequences were seen quickly (90% within the first 25% of the trace).

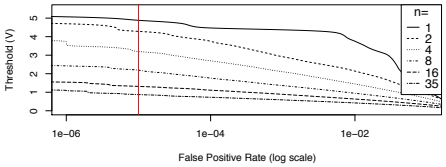

Fig. 7. For a given false positive rate and community size, we can compute the threshold V. The vertical red line, for instance, corresponds to about one false positive per six days.

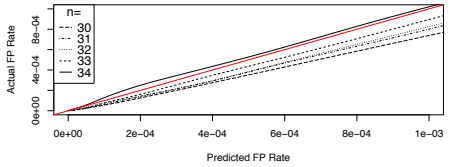

Fig. 8. The training data is a good predictor of the false positive rates seen in monitoring data. The threshold V can be set as high as necessary to achieve an acceptable rate of false positives.

The fact that training speeds up with community size is consistent with previous work [21]; Syzygy's distinctive use of the community occurs during the monitoring phase (Section 5.3).

During this training period, while the clients were reporting both the complete sequences and timestamps at an average of 100 KB/s, the average bandwidth usage at the server was 1160 KB/s (the peak was 3240 KB/s). The clients required less than 1% CPU each for the `strace` process and Syzygy script. With all 35 clients active, the server-side script was using 13% of the processor, on average, with peaks as high as 32%.

Even though the training data includes machines that are unlike most of the cluster, such as two laptops, we still find that the distribution of community anomaly scores within the training community converges toward a tight normal distribution. Figure 6 shows the standard deviation of the community score for increasing numbers of clients; in the figure, the clients "join" the community in reverse order of average anomaly score (so $n = 1$ represents the client with the highest average anomaly score). To evaluate the impact of heterogeneity, we also plot four hypothetical communities: "Outliers Removed," where the two laptops and the Syzygy server were replaced with the client with the lowest standard deviation, "High SD Removed," where the five clients with the highest standard deviations were replaced with five clones of the machine with the lowest standard deviation, and "Homogeneous" and "Homogeneous (High SD)," which are communities of n clones of the client with the lowest average anomaly score and highest standard deviation, respectively. The results show that variance in the community score comes not from client heterogeneity (the client in "Homogeneous (High SD)" was a normal cluster machine) but from client variance. The results also show that a larger community can compensate for client variance.

Section 3.3 shows how to compute the threshold V, given a desired false positive rate and the training data; these analytical results correspond well with what we observe experimentally. Using the data from our deployment, Figure 7 plots the appropriate choice of V for a desired false positive rate (note the log scale) and community size (n). The units of the false positive rate, for this deployment, are expected false positives per five seconds. The vertical line is a hypothetical target rate: 1×10^{-5} (about six days). The y-value at which this line intercepts each community size line is the threshold for that value of n.

5.3 Distributed Monitoring

After training is complete, Syzygy switches to monitoring mode. For these experiments, we set $T_i = \infty$ to prevent hiatons from being introduced. (We omit the exploration of T_i values for space reasons.) Over the course of a week, we collected just under 10 billion anomaly scores from the community. Five clients seen during training were not heard from again, while four new ones appeared. There were no epidemics nor other coordinated events during the monitoring period; the machines are part of the campus computing infrastructure, so we could not obtain permission to stage an epidemic.

The `strace` process on the client requires an average of 1–2% CPU overhead, and the Syzygy client script requires another 2–3% to calculate the anomaly scores and send them to the server. The server-side Syzygy process uses less than 1% of the CPU for a single client; our experiments suggest a server could easily handle more than a hundred clients (see Section 7).

Syzygy can either send one packet per anomaly score or buffer some number before reporting them. At an average rate of 2000 system calls per second, sending one packet per call would be inefficient. Buffering 100 scores with a short timeout to ensure freshness, for example, reduces the bandwidth requirements to 20 packets per second at 1.5 KB per packet ($\sim$ 30 KB/s), including the overhead of transmitting timestamps along with the anomaly scores, which we did for experimental purposes. Communicating the scores alone would require less than half this bandwidth.

Section 3.3 notes that achieving the target false positive rate requires that μ_X and σ_X accurately describe the future distribution of anomaly scores. Figure 8 quantifies that statement using the deployment data collected while Syzygy was in monitoring mode (data not used to build the model). The diagonal red line indicates perfect agreement. Even at very low false positive rates and small community sizes, the modeling data was sufficient to allow good prediction of the false positive rate on real monitoring data.

6 Controlled Experiments

In this section, we test Syzygy in a controlled environment under various adverse conditions, using trace data from commodity applications and exploits capable of sophisticated behaviors.

An *experiment* is a binary classification problem in which Syzygy is given a sequence of anomaly scores for n clients and must decide whether 0 of them are infected (healthy) or whether $d \geq 1$ of them have been exploited (infected). Thus, an *example* is a set of n score vectors of length W_i. Ideally, Syzygy should report an epidemic iff one or more of the score vectors was produced by an infected client. We use standard metrics to evaluate performance on this classification problem: false positive rate (FP), false negative rate (FN), true positive rate (TP), true negative rate (TN), and F1 Measure ($\frac{2TP}{2TP+FP+FN}$), which combines precision and recall, weighting each equally.

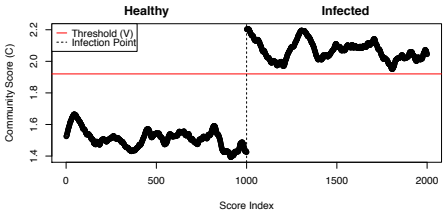

Fig. 9. A pair of examples, using Camino and the showpages exploit with $n = 100$ and $d = 5$, showing a TN and a TP.

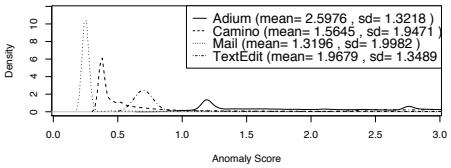

Fig. 10. Healthy anomaly distributions, plotted with a kernel density estimator. The bump at around 2.75 suggests Adium's model is imperfect.

For example, say we are measuring Syzygy's performance on a community of size $n = 100$ and epidemic of size $d = 5$. We produce an example of an infected community as follows. Say that we have already constructed models for all n clients and have the associated system call traces. To construct each of the $n - d$ healthy score vectors, we pick a window from the application trace, uniformly at random, and compute the anomaly scores as described in Section 4.1. (The sample window determines R_i.) Using exploit traces, we construct d infected score vectors. Syzygy then takes the n vectors of anomaly scores and computes the elementwise averages. If $C > V$ for *any* element C of the resulting community score vector, then Syzygy classifies the example as infected; otherwise, it classifies it as healthy. Using data described in Section 6.1, we plot the community scores for a pair of examples in Figure 9; a healthy example is on the left and an infected example on the right. In other words, in the plot, the first 1000 scores are from a healthy community, while the next 1000 are from an infected community—Syzygy classifies them based on V, reporting an epidemic when it sees the first score from the infected community.

We repeat this randomized process 1000 times per example to get statistically meaningful metrics. We always present Syzygy with an equal number of healthy and infected examples, though Syzygy does not use this fact in any way. This is not meant to reflect the base rate of intrusions in a system, but increases the precision of the metrics. As the size of the training set goes to infinity, it becomes irrelevant as to whether or not we remove the current trace file from the training set because its influence goes to zero. It is sufficient to select random windows from the traces because Syzygy is memoryless outside of each sample. Unless noted otherwise, we set $W_i = 1000$ sequences and $V = \mu_H + 2\sigma_H$, where H is the distribution of community scores for a community of size n, as in Section 3.3. We present the results of our controlled experiments in Sections 6.2–6.5.

6.1 Data

We collect system call and timing traces from commercial, off-the-shelf software under normal usage by the authors, using the utility `dtrace`. We use several desktop applications: a chat program (Adium), a web browser (Camino), a mail client (Mail), and a simple text editor (TextEdit). A summary of these data

Application	**Version**	**Calls**	**Time** (sec)	**Rate** (calls/sec)	**Unique**	T_i (sec)
Adium	1.2.7	6,595,834	33,278	198.204	50,514	54.451
Camino	1.6.1Int-v2	113,341,557	57,385	1975.11	103,634	7.2605
Mail	3.3	106,774,240	48,630	2195.65	126,467	896.85
TextEdit	1.5 (244)	176,170	31,794	5.54098	4469	6031.4

Table 2. Training data. The Unique column indicates the number of unique length-six sequences. T_i is the maximum time from the beginning of one system call to the start of the next.

is provided in Table 2. When compared to the real deployments in Sections 4 and 5, we find that our simulations are a reasonable approximation. Note that, although Syzygy must build a dynamic model of application behavior, it does not need to learn exploit signatures.

Many exploits currently found in the wild are brazen about their misbehavior (large δ) and are therefore easy for Syzygy to detect (see Section 3.3). Instead, we focus in this section on Syzygy's distinguishing ability to detect next-generation exploits under adverse conditions. These exploits can infect the application at any execution point (i.e., multiple infection vectors), are privy to all of Syzygy's data and parameters, and can perform skillful mimicry. The adverse conditions include client heterogeneity and tainted training data.

In order to simulate such behavior, we use four next-generation exploits: *mailspam* infects Mail, then composes and sends a large number of emails (based on the open mail relay in the Sobig worm's trojan payload); *prompttext* infects TextEdit, then asks the user for input that it writes to a file (based on file creation and deletion seen in SirCam, Chernobyl, or Klez [40]); *screenshot* infects Adium, then takes a snapshot of the current display (like prompttext but without user interaction); and *showpages* infects Camino, then loads a series of web pages (based on HTML proxies like Sobig's trojan, DoS payloads like Code Red and Yaha, and self-updating payloads like W32/sonic and W32/hybris [6]).

Except where noted, we gathered data using an Apple MacPro with two 2.66 GHz Dual-Core Intel Xeons and 6 GB of memory running Mac OS X 10.5.4, and the results we present are representative. Using the resulting model, we compute the distribution X of healthy client anomaly scores for each program (Figure 10). The results of Section 5.2 show that behavioral variance comes from client behavior over time, rather than client heterogeneity; the smartest way to gather a good data set was, therefore, to monitor a single client for a long time. Section 6.4 provides experiments supporting the merit of that decision.

We use the phrase "normal usage" to mean that no artificial workloads were generated nor were certain activities prescribed. As is evident from the rate of new sequences seen during training, plotted in Figure 11, we made no effort to train until convergence, nor to exercise the rarer features of these programs. We also do not separate sequences by thread, instead ordering them strictly by time of invocation. The resulting models are therefore small, imprecise, and

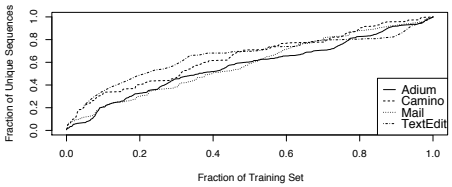

Fig. 11. The applications generate new sequences throughout training, with occasional bursts (e.g., program launches).

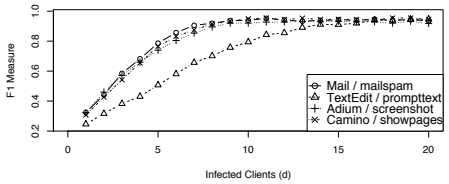

Fig. 12. F1 measure with $n = 100$ and varying infection size (d) using each of the four pairs of programs and exploits.

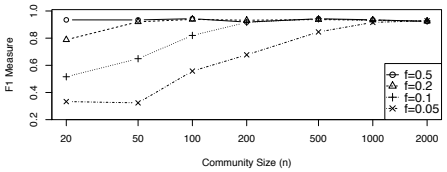

Fig. 13. F1 measure with varying community size and constant fraction $f = d/n$ infected, using TextEdit and prompttext.

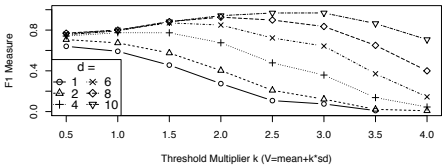

Fig. 14. F1 measure with $n = 100$ and varying threshold multiplier using traces from Mail and the mailspam exploit.

incomplete, as we might expect to achieve in practice; the Syzygy performance numbers we present would only improve with better models.

6.2 Detection Performance

We first consider Syzygy's ability to detect epidemics for various sizes of community and infected population. Consider the experiments plotted in Figure 12 wherein a fixed-size community is being infected. Syzygy's performance improves with infection size, peaking, in this experiment, at around 10 exploited clients (10% of the community). Figure 13 shows, however, that with a sufficiently large community we require a vanishingly small fraction of the population to be sacrificed before we detect the exploit. Although the community and infected population are growing at the same rate, Syzygy's ability to detect the infection outpaces that growth.

6.3 Parameter Sensitivity

We next evaluate Syzygy's sensitivity to the threshold V. Figure 14 shows performance for various choices of V. Once the community and infected population are sufficiently large, we see the performance curve reach a maximum at a point between $V = \mu_X$ and μ_Y. Increasing the multiplier tends to increase precision, decrease recall, and decrease the false positive rate (which falls off like the tail of the normal distribution). To further visualize this, see Figure 15. As the number of clients grows, the normal and infected distributions become more clearly separated. This increasing noise margin suggests that the exact placement of the

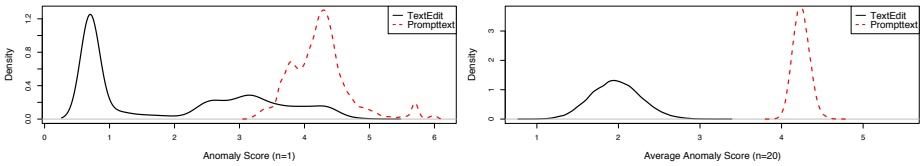

Fig. 15. The left plot shows anomaly signal density estimates for TextEdit and the prompttext exploit. There is no ideal position on the x-axis to set a threshold. On the right, we see that averaging scores across a number of clients yields a clearer separation.

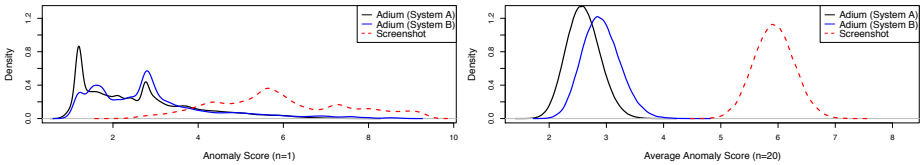

Fig. 16. Similar to Figure 15, except using the Adium program and giving data for both our primary system (System A) and the laptop (System B). All curves are based on the Adium model built using only System A.

threshold does not strongly affect Syzygy's performance. Indeed, in the limit, all choices of threshold $\mu_X < V < \mu_Y$ yield perfect detection.

6.4 Client Variation

We expect clients to differ in machine specifications and configurations, and for these to change over time. To test this situation, we ran the same applications as on our primary test machine (*System A*) on a second system (*System B*) with different specifications: an Apple PowerBook G4 with a single 1.33 GHz PowerPC processor and 1.25 GB of memory running Mac OS X 10.5.4. The data is summarized in Table 3. In Figure 16, we compare the anomaly scores for these Adium traces against those from the training system and the screenshot exploit. Although System B's average score is higher by Δ (its model is from another system), the programs behave similarly enough on both systems that unusual but healthy clients are not easily confused with exploits.

As the community grows, however, System B begins looking like an exploit. The healthy community score distribution variance, σ_H, shrinks, so V moves closer to μ_X, slowly passing below System B's average anomaly score. This contrived problem is easily remedied by using a model constructed from System B's behavior rather than System A's, or by normalizing the anomaly scores from System B as prescribed in Section 3.2. In practice, such a situation may arise when a client upgrades the application but does not retrain the model; if a client's anomaly signal remains high for long periods of time, this may indicate that the model is no longer valid—only when many clients make such changes would we expect spurious epidemic reports. Section 5 contains additional results related to client variation that suggest heterogeneity is not a problem in practice.

Program	**Version**	**Time** (sec)	**Rate** (calls/sec)	**Unique**	T_i (sec)	$\approx \Delta$
Adium	1.2.7	2093	54.8839	6749	47.457	0.31589
Camino	1.6.1Int-v2	3901	868.294	21,619	1.84077	0.60442
Mail	3.3 (926.1/926)	1126	16.2869	7963	421.645	0.53272
TextEdit	1.5 (244)	2506	92.8204	2925	528.164	1.17758

Table 3. Data from OS X apps on a different client. The Unique column indicates the number of unique length-six sequences, and T_i is the maximum time from the beginning of one system call to the start of the next. The Δ column shows the empirically estimated average difference between anomaly scores on Systems A and B.

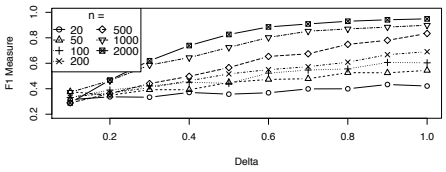

Fig. 17. Varying δ using Adium, with $d/n = 0.1$. Mimicry makes detection more difficult, but, at higher δs, performance improves logarithmically with n.

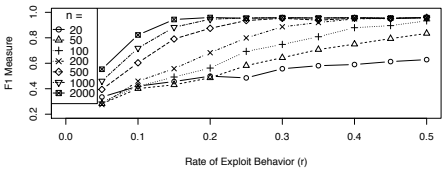

Fig. 18. Varying rate of bad behavior (r) using Camino and showpages, with $d/n = 0.1$. A sufficiently large community guarantees that bad behavior will overlap.

6.5 Mimicry and Tainting

An exploit can avoid detection if its behavior is sufficiently similar to the application's, from the perspective of a given model [38]. There are two ways an exploit might mimic application behavior: (i) by ensuring that the distribution of anomaly scores is sufficiently similar or (ii) by limiting the *rate* at which it exhibits bad behavior. Perfect mimicry, in which exploit behavior is indistinguishable from application behavior, can never be detected, by definition, using any behavior-based epidemic detector; however, we can show Syzygy is robust against a very high degree of mimicry and against rate-limiting an attack.

Scenario (i), mimicking the distribution, is quantified in Syzygy by the parameter δ. Recall that a lower value for δ means the two distributions are more similar. Tainted training data is symmetric to mimicry: raising μ_X instead of lowering μ_Y. Either way, δ is decreased and the following results hold. Intuitively, these experiments simulate an exploit that makes system call sequences in similar (but not identical) proportions to the application. This is done computationally by generating anomaly scores from the application's distribution, then shifting them positively by δ. ($Y \sim X + \delta$.)

Figure 17 gives results from these experiments. Syzygy is able to detect fairly well even for low δ. The poor performance at the lowest δs, despite large communities, is almost exclusively a result of false negatives: V is set too high. With a lower V, we can get F1 > 0.6 even when $\delta = 0.1$, $n = 10$, and $d = 1$.

We now consider scenario (ii), limiting bad behavior to a fixed rate. Specifically, if the exploit spreads bad behavior out over time, in bursts that cumulatively account for a fraction r of the runtime per client, such that the community

signal does not deviate above $\mu_X + V$, no epidemic will be reported. Mathematically, this attack corresponds to decreasing the effective infection size from d to dr. This, in itself, may be considered a victory under certain circumstances, such as when a worm may be contained so long as it does not spread too quickly [42]. In our experiment, we splice windows of infected anomaly scores into sequences of healthy anomaly scores, in proportions determined by the rate r. Figure 18 shows how Syzygy performs against this rate-limiting attack. Again, false negatives dominate the metric—with a better-chosen V, we can get F1 above 0.68 at $r = 0.05$ with as few as 10 clients.

7 Scalability

Mathematically, Syzygy's accuracy improves as the community grows, so it is crucial that the implementation scales well. This issue is independent of the analysis in Section 3. We described the infrastructure as using a central server, and demonstrated that it works for as many as 35 clients (Section 5). Communication is one-way (client to server) and there is no consensus or agreement protocol, so the total community traffic scales linearly with the number of clients.

This central server may be replaced, however, with alternatives that would increase scalability and avoid a single point of failure. One option is a server hierarchy; each server computes the community score for its children and reports this value and the size of that sub-community to a parent server. This arrangement works precisely because the function used to compute the community score, `mean()`, is associative (when weighted by sub-community size).

In addition to communication overhead, there is monitoring overhead on the clients. This is typically a consequence of model choice and unaffected by community size. In our controlled experiments, the primary monitoring tool, `dtrace`, required less than 10% of one CPU even during heavy activity by the monitored application; the average usage was below 1%. In our deployment experiments with Firefox, Syzygy required less than 5% of the CPU on average, and 7% peak, including `strace` overhead (see Section 5.3). Using our `strace`-based implementation for Windows, however, the slowdown was noticeable. The overhead in our Apache deployment (see Section 4), which took advantage of the web server's built-in logging mechanism, was negligible. If overhead becomes problematic, then it may be worth changing the model to measure less costly signals. For example, Sharif et al [30] implemented control-flow monitoring with overhead comparable to our system call-based approach—this optimization would likely yield greater precision at lower overhead.

8 Contributions

Syzygy is an epidemic detection framework that looks for time-correlated anomalies in a homogeneous software community—precisely the behavior that would accompany an exploit as it executes among a set of clients. Our results show that Syzygy is effective at automated detection of epidemics, is practical to deploy, and scales well. Syzygy takes advantage of the statistical properties of large communities in a novel way, asymptotically approaching perfect detection.

Acknowledgments

The authors thank the members of the VERNIER team, especially Elizabeth Stinson, Patrick Lincoln, Steve Dawson, Linda Briesemeister, Jim Thornton, John Mitchell, and Peter Kwan. Thanks to Sebastian Gutierrez and Miles Davis for help deploying Syzygy, to Naeim Semsarilar for his invaluable contributions to the early stages of this work, and to Xuân Vũ for her input and support.

References

[1] A. Bouloutas, S. Calo, and A. Finkel. Alarm correlation and fault identification in communication networks. In *IEEE Transactions on Communications*, 1994.
[2] D. Brumley, J. Newsome, and D. Song. Sting: An end-to-end self-healing system for defending against internet worms. *Malware Detection and Defense*, 2007.
[3] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *SOSP*, 2005.
[4] F. Cuppens and A. Miege. Alert correlation in a cooperative intrusion detection framework. In *IEEE Symposium on Security and Privacy*, pages 202–215, 2002.
[5] H. Debar, M. Becker, and D. Siboni. A neural network component for an intrusion detection system. In *IEEE Symposium on Security and Privacy*, 1992.
[6] D. Ellis. Worm anatomy and model. In *WORM*, 2003.
[7] E. Eskin. Anomaly detection over noisy data using learned probability distributions. In *ICML*, 2000.
[8] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, 2003.
[9] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*, 1996.
[10] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *CCS*, 2004.
[11] D. Gao, M. K. Reiter, and D. Song. Behavioral distance for intrusion detection. In *RAID*, 2006.
[12] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *USENIX Security*, pages 61–79, 2002.
[13] G. Gu, A. A. Cárdenas, and W. Lee. Principled reasoning and practical applications of alert fusion in intrusion detection systems. In *ASIACCS*, 2008.
[14] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
[15] L. Huang, M. Garofalakis, A. D. Joseph, and N. Taft. Communication-efficient tracking of distributed cumulative triggers. In *Intl. Conf. on Distributed Computing Systems (ICDCS)*, June 2007.
[16] L. Huang, X. L. Nguyen, M. Garofalakis, J. Hellerstein, M. Jordan, A. Joseph, and N. Taft. Communication-efficient online detection of network-wide anomalies. In *IEEE INFOCOM*, 2007.
[17] G. Jakobson and M. Weissman. Alarm correlation. In *IEEE Network*, 1993.
[18] H. S. Javitz and A. Valdes. The SRI IDES statistical anomaly detector. In *IEEE Symposium on Security and Privacy*, 1991.
[19] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Constructing attack scenarios through correlation of intrusion alerts. In *CCS*, 2002.
[20] P. Lincoln, et al. Virtualized Execution Realizing Network Infrastructures Enhancing Reliability (VERNIER). `http://www.sdl.sri.com/projects/vernier/`.

[21] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software self-healing using collaborative application communities. In *NDSS*, 2005.
[22] D. J. Malan and M. D. Smith. Host-based detection of worms through peer-to-peer cooperation. In *ACM Workshop on Rapid Malcode*, 2005.
[23] D. J. Malan and M. D. Smith. Exploiting temporal consistency to reduce false positives in host-based, collaborative detection of worms. In *WORM*, 2006.
[24] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. In *TISSEC*, 2006.
[25] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, 2006.
[26] P. Ning, Y. Cui, and D. S. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *CCS*, 2002.
[27] V. Paxson. Bro: a system for detecting network intruders in real-time. In *Computer Networks*, volume 31, 1999.
[28] P. A. Porras and P. G. Neumann. Emerald: event monitoring enabling responses to anomalous live disturbances. In *National Computer Security Conference (NIST/NCSC)*, 1997.
[29] M. M. Sebring and R. A. Whitehurst. Expert systems in intrusion detection: a case study. In *National Computer Security Conference*, 1988.
[30] M. Sharif, K. Singh, J. Giffin, and W. Lee. Understanding precision in host based intrusion detection. In *RAID*, 2007.
[31] S. Smaha. Haystack: an intrusion detection system. In *Aerospace Computer Security Applications Conference*, 1988.
[32] S. Staniford-chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagl, K. Levitt, C. Wee, R. Yip, and D. Zerkle. Grids—a graph based intrusion detection system for large networks. In *NIST/NCSC*, 1996.
[33] K. M. C. Tan and R. A. Maxion. "Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector. In *IEEE Symposium on Security and Privacy*, 2002.
[34] J. Ullrich. `http://www.dshield.org`. DShield—distributed intrusion detection system.
[35] H. Vaccaro and G. Liepins. Detection of anomalous computer session activity. In *IEEE Symposium on Security and Privacy*, 1989.
[36] A. Valdes and K. Skinner. Probabilistic alert correlation. In *RAID*, 2001.
[37] W. W. Wadge and E. A. Ashcroft. Lucid, the dataflow programming language. *A.P.I.C. Studies in Data Processing*, 1985.
[38] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS*, 2002.
[39] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.
[40] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy of computer worms. In *WORM*, 2003.
[41] N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *USENIX Security*, 2004.
[42] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *ACSAC*, 2002.
[43] Y. Xie, H. Kim, D. O'Hallaron, M. Reiter, and H. Zhang. Seurat: a pointillist approach to anomaly detection. In *RAID*, September 2004.
[44] V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the DOMINO overlay system. In *NDSS*, 2004.