# Automated Response Using System-Call Delays


Anil Somayaji
*Dept. of Computer Science*
*University of New Mexico*
*Albuquerque, NM 87131*
soma@cs.unm.edu

Stephanie Forrest
*Santa Fe Institute*
*Santa Fe, NM 87501*
*Dept. of Computer Science*
*University of New Mexico*
*Albuquerque, NM 87131*
steph@santafe.edu, forrest@cs.unm.edu



## Abstract

Automated intrusion response is an important unsolved problem in computer security. A system called pH (for process homeostasis) is described which can successfully detect and stop intrusions before the target system is compromised. In its current form, pH monitors every executing process on a computer at the system-call level, and responds to anomalies by either delaying or aborting system calls. The paper presents the rationale for pH, its design and implementation, and a set of initial experimental results.


## 1 Introduction

This paper addresses a largely ignored aspect of computer security—the automated response problem. Previously, computer security research has focused almost entirely on prevention (e.g., cryptography, firewalls and protocol design) and detection (e.g., virus and intrusion detection). Response has been an afterthought, generally restricted to increased logging and administrator email. Commercial intrusion detection systems (IDSs) are capable of terminating connections, killing processes, and even blocking messages from entire networks [3, 12, 22]; in practice, though, these mechanisms cannot be widely deployed because the risk of an inappropriate response (e.g., removing a legitimate user's computer from the network) is too high. Thus, IDSs become burdens, requiring administrators to analyze and respond to almost every detected anomaly. In an era of expanding connectivity and ubiquitous computing, we must seek solutions that reduce the system administrator's workload, rather than increasing it. That is, our computers must respond to attacks autonomously.


Published in the Proceedings of the 9th USENIX Security Symposium (August 14–17, 2000).



In earlier work, we and others have demonstrated several methods of anomaly detection by which large classes of intrusions can be detected, e.g., [1, 27, 17, 16]. Good anomaly detection, however, comes at the price of persistent false positives. Although more sophisticated methods will no doubt continue to be developed, we believe that it is infeasible to eliminate false positives completely. There are several reasons for this. First, computers live in rich dynamic environments, where inevitably there are new patterns of legitimate activity not previously seen by the system — a phenomenon known as *perpetual novelty* (see Hofmeyr [21] for an empirical model of the rate at which new patterns appear in a local area network). Second, profiles of legitimate activity change continually, as computers and users are added or deleted, new software packages or patches are added to a system, and so forth. Thus, the normal state of the system is *evolving* over time. Finally, there is inherent ambiguity in the distinction between normal and intrusive (or abnormal) activities. For example, changes to system configuration files are legitimate if performed by a system administrator; however, the very same actions are a security violation conducted done by a non-privileged user or an outside attacker. Thus, any automated response system must be designed to account for persistent false-positives, evolving definitions of normal, and ambiguity about what constitutes an anomaly.

We have chosen to focus on automated response mechanisms which will allow a computer to preserve its own integrity (i.e. stay "alive" and uncompromised), rather than ones that help discover the source or method of an intrusion. Within this context, we believe that the best way to approach the automated response problem is by designing a system in which a computer autonomously monitors its own activities, routinely making small corrections to maintain itself in a "normal" state. In biology, the maintenance of a stable (normal) internal environment is known as *homeostasis*. All living systems employ a wide range of homeostatic mechanisms in order to survive under fluctuating environmental conditions. We propose that computer systems should similarly have mechanisms which strive to maintain a stable environment inside the computer, even in the face of wide variations in inputs. Under this view, automated response is recast from a monolithic all-or-nothing action (which if incorrect can have dire consequences) to a set of small, continually occurring changes to the state of the system. With this view, occasional false alarms are not problematic, because they have small impact. In earlier papers, we have advocated a view of computer security based on ideas from immunology [16, 34, 20]. This paper naturally extends that view by recognizing that immune systems are more properly thought of as homeostatic mechanisms than pure defense mechanisms [26].

In the following sections, we describe a working implementation of these ideas—a set of extensions to a Linux kernel which does not interfere with normal operation but can successfully stop attacks as they occur. We call the system pH (short for process homeostasis). To create pH, we extended our earlier intrusion-detection work using system calls [16] by connecting system calls with feedback mechanisms that either delay or abort anomalous system calls.

Delays form a natural basis for interfering with program behavior: small delays are typically imperceptible to a program, and are minor annoyances to a user. Longer delays, however, can trigger timeouts at the application and network levels, effectively terminating the delayed program. By implementing the delays as an increasing function of the number of recent anomalous sequences, pH can smoothly transition between normal execution and program termination.

This paper makes two principal contributions. First, it demonstrates the feasibility of monitoring every active process at the system-call level in real-time, with minimal impact on overall performance. Second, it introduces a practical, relatively non-intrusive method for automatically responding to anomalous program behavior.

The paper proceeds as follows. First, we review our system call monitoring and anomaly detection method. Next, we explain the design and implementation of pH. We then demonstrate pH's effectiveness at stopping attacks, show through benchmarks that it runs with low overhead, and describe what it is like to actually use pH on a workstation. After a review of related work, we conclude with a discussion of limitations and future work.

## 2 Background

Both the monitoring and the response components of pH use ideas introduced in [16]. What follows is a description of our original testing methodology, with which we gathered on-line data for off-line analysis. Subsequent sections explain how these techniques were modified to create pH.

To review, we monitored all the system calls (without arguments) made by an executing program on a per-process basis. That is, each time a process was invoked, we began a new trace, logging all the system calls for that process. Thus, for every process the trace consists of an ordered list (a time-series) of the system calls it made during its execution. For commonly executed programs, especially those that run with privilege, we collected such traces over many invocations of the program, when it was behaving normally. We then used the collection of all such traces (for one program) to develop an empirical model of its normal behavior.

Once the system had been trained on a sufficient number of normal program executions, the model was tested on subsequent invocations of the program. The hope was that the model would recognize most normal behavior as "normal" and most attacks as "abnormal." Our method thus falls into the category of anomaly intrusion detection.

Given a collection of system call traces, how do we use them to construct a model? This is an active area of research in the field of machine learning, and there are literally hundreds of good methods available to choose from, including hidden Markov models, decision trees, neural networks, and a variety of methods based on deterministic finite automata (DFAs). We chose the simplest method we could think of within the following constraints. First, the method must be suitable for on-line training and testing. That is, we must be able to construct the model "on the fly" in one pass over the data, and both training and testing must be efficient enough to be performed in real-time. Next, the method must be suitable for large alphabet sizes. Our alphabet consists of all the different system calls—typically about 200 for UNIX systems. Finally, the method must create models that are sensitive to common forms of intrusion. Traces of intrusions are often 99% the same as normal traces, with very small, temporally clumped deviations from normal behavior. In the following, we describe a simple method, which we call "time-delay embedding" [16]. Warrender [38] compared time-delay embedding with several other common machine learning algorithms and discovered that it is remarkably accurate and efficient in this domain.

We define normal behavior in terms of short $n$-grams of system calls. Conceptually, we define a small fixed size window and "slide" it over each trace, recording which calls precede the current call within the sliding window. The current call and a call at a fixed preceding window position form a "pair," with the contents of a window of length $x$ being represented by $x-1$ pairs. The collection of unique pairs over all the traces for a single program constitutes our model of normal behavior for the program.[1]

More formally, let

$$
\begin{aligned}
S &= \text{alphabet of possible system calls} \\
T &= \text{trace} \\
&= \text{the sequence } t_0, t_1, \ldots, t_{\tau-1}, t_i \in S \\
w &= \text{window size, } 2 \leq w \leq \tau \\
P &= \text{profile} \\
&= \text{set of patterns associated with } T \text{ and } w \\
&= \{\langle s_i, s_j \rangle_k : s_i, s_j \in S, 1 \leq k < w \\
&\qquad\qquad \exists p: \quad 0 \leq p < \tau - k, \\
&\qquad\qquad\qquad t_p = s_i, \\
&\qquad\qquad\qquad t_{p+k} = s_j,
\end{aligned}
$$

For example, suppose we had as normal the following sequence of calls:

execve, brk, open, fstat, mmap, close, open, mmap, munmap

and a window size of 4. We slide the window across the sequence, and for each call we encounter, we record what call precedes it at different positions within the window, numbering them from 0 to $w-1$, with 0 being the current system call. So, for this trace, we get the following windows:

| position 3 | position 2 | position 1 | current |
|---|---|---|---|
| | | | execve |
| | | execve | brk |
| | execve | brk | open |
| execve | brk | open | fstat |
| brk | open | fstat | mmap |
| open | fstat | mmap | close |
| fstat | mmap | close | open |
| mmap | close | open | mmap |
| close | open | mmap | munmap |

When a call occurs more than once in a trace, it will likely be preceded by different calls in different contexts. We compress the explicit window representation by joining together lines with the same current value (note the open and mmap rows):

| current | position 1 | position 2 | position 3 |
|---|---|---|---|
| execve | | | |
| brk | execve | | |
| open | brk, close | execve, mmap | fstat |
| fstat | open | brk | execve |
| mmap | fstat, open | open, close | brk, mmap |
| close | mmap | fstat | open |
| munmap | mmap | open | close |

This table can be stored using a fixed-size bit array. If $|S|$ is the size of the alphabet (number of different possible system calls) and $w$ is the window size, then we can store the complete model in a bit array of size: $|S| \times |S| \times (w-1)$. Because $w$ is small (6 is our standard default), our current implementation uses a $200 \times 200$ byte array, with masks to access the individual bits.

At testing time, system call pairs from test traces are compared against those in the normal profile. Any system call pair (the current call and a preceding call within the current window) not present in the normal profile is called a *mismatch*. Any individual mismatch could indicate anomalous behavior (a true positive), or it could be a sequence that was not included in the normal training data (a false positive). The current system call is defined as anomalous if there are any mismatches within its window.

To date, all of the intrusions we have studied produce anomalous sequences in temporally local clusters. To facilitate the detection of these clusters, we record recent anomalous system calls in a fixed-size circular array, which we refer to as a *locality frame*. More precisely, let $n$ be the size of our locality frame, and let $A_i$ be the $i$-th entry of the locality frame array, with $0 \leq i < n$ and $A_i \in \{0, 1\}$. Then, for system call $s$ ($0 \leq s < \tau$) with mismatches $m_s$, $A_{s \bmod n} = 1$ iff $m_s > 0$, and is 0 other-

[1] Our original paper on using system calls for intrusion detection [16] used a technique called "lookahead pairs." pH uses the original lookahead pairs algorithm as described here, except that it looks behind instead of ahead. Later papers [20, 38] report results based on recording full sequences. We reverted to lookahead pairs because it is simple to implement and extremely efficient.

wise. Thus, the locality frame implicitly stores the number of the past $n$ system calls which were anomalous. We call this total of recent anomalies, $\sum A_i$, the locality frame count (LFC).[2] For the experiments described below, we used a locality frame of size 128.

## 3 pH Design

pH performs two important functions: It monitors individual processes at the system-call level, and it automatically responds to anomalous behavior by either slowing down or aborting system calls. Normal behavior is determined by the currently running binary program; response, however, is determined on a per-process basis.

To minimize I/O requirements and maximize efficiency, stability, and security, we have implemented most of pH in kernel space. We considered several alternative approaches, including audit packages, system-call tracing utilities (such as `strace`), and instrumented libraries. However, each of these other approaches has serious drawbacks. Audit packages generate voluminous logfiles, which are expensive to create and even more expensive to analyze. Additionally, they do not routinely record every system call. User-space tracing utilities are too slow for our application, and in some cases, they interfere with privileged daemons to the extent that they behave incorrectly. Instrumented libraries cannot detect every system call, because not every system call comes through a library function (e.g., buffer overflow attacks). In addition, a kernel implementation allows us to put our monitoring and response mechanisms exactly where they are needed, in the system call dispatcher, and allows the implementation to be as secure as the kernel.

For each running executable, pH maintains two arrays of pair data: A training array and a testing array. The training array is continuously updated with new pairs as they appear; the testing array is used to detect anomalies, and is never modified except by replacing it with a copy of the training array. Put another way, the testing array is the current normal profile for a program, while the training array is a candidate future normal profile.

A new "normal" is installed by replacing the testing array with the current state of the training array. The replacement occurs under three conditions: (1) the user explicitly signals via a special system call (`sys_pH`) that a profile's training data is valid; (2) the profile anomaly count exceeds the parameter $anomaly\_limit$; (3) the training formula is satisfied. When an anomaly is detected, the current system call is delayed according to a simple formula. Details of these conditions and actions are given in the next several paragraphs.

The training to testing copy can occur automatically based on the state of the following training statistics:

$$\begin{aligned} train\_count: &\quad \text{\# calls since array initialization} \\ last\_mod\_count: &\quad \text{\# calls since array was last modified} \\ normal\_count = &\quad train\_count - last\_mod\_count \end{aligned}$$

When the training array meets all of the following conditions, it is copied onto the testing array (note: this is the normal mechanism for initiating anomaly detection in the system):

$$\begin{aligned} last\_mod\_count &> mod\_minimum \\ normal\_count &> normal\_minimum \\ \frac{train\_count}{normal\_count} &> normal\_ratio \end{aligned}$$

The three parameters on the right are user defined, and can be set at runtime.

As we mentioned earlier, pH responds to anomalies by delaying system call execution. The amount of delay is an exponential function of the current LFC, regardless of whether the current call is anomalous or not. The unscaled delay for a system call is $d = 2^{\text{LFC}}$. The effective delay for a system call is $d \times delay\_factor$, where $delay\_factor$ is another user-defined parameter. Note that delays may be disabled by setting $delay\_factor$ to 0. If the LFC ever exceeds the $tolerization\_limit$ parameter (which is 12 for the experiments described below), the training array is reset, preventing truly anomalous behavior from being incorporated into the testing array.

Because pH monitors process behavior based on the executable that is currently running, the execve system call causes a new profile to be loaded. Thus, if an attacker were able to subvert a process and cause it to make an execve call, pH might be tricked into treating the current process as normal, based on the data for the newly-loaded executable. To avoid this possibility the maximum LFC count (maxLFC) for a process is recorded. If

[2] A somewhat different approach was taken in Hofmeyr [20], where the measure of anomalous behavior was based on Hamming distances between unknown sequences and their closest match in the normal database.

maxLFC exceeds the $abort\_execve$ threshold, then all execve's are aborted for the anomalous process.

pH also keeps a count of the raw number of anomalies each profile has seen. This count can be seen as a measure of ongoing, non-clustered abnormal behavior. If this number exceeds the parameter $anomaly\_limit$, pH automatically copies the training array to the testing array, causing pH to treat similar future behavior as normal. Borrowing from immunology, we refer to this process as *tolerization*. Low values of $anomaly\_limit$ allow pH to automatically tolerize most novel behavior, while higher values inhibit tolerization. When a system is initially set up, automatically-created normal profiles may contain too little normal behavior. To reduce the number of reported anomalies, $anomaly\_limit$ should be set to a small value (less than 10). Then, once the system has stabilized, $anomaly\_limit$ should be set to at least 20 to prevent pH from automatically learning the behavior of attacks.

## 4 Implementation

The pH prototype is implemented as a patch for the Linux 2.2 kernel, and was developed and tested on systems running a pre-release of the Debian/GNU Linux 2.2 distribution [35]. The modified kernel is capable of monitoring every executed system call, recording profiles for every executable. An overview of the system is shown in Figure 1.

Program profiles for each executable are stored on disk. Each profile contains both a training and testing array, and so is actually two "profiles" by the terminology in Section 2. The kernel loads the current profile when a new program begins executing (on execve), and then writes it out again when the process terminates. When a new executable is loaded via the execve system call, the kernel attempts to load the appropriate profile from disk; if it is not present, a new profile is created. If another process runs the same executable, the profile is shared between both processes. To prevent consistency problems due to interleaving, each executing process maintains its own record of recent system calls (its current sequence). When all processes using a given profile terminate, the updated profile is saved to disk. A loaded profile consumes approximately 80K of kernel (non-swappable) memory.

We modified the system call dispatcher so that it calls a pH function (`pH_process_syscall`) prior to dispatching the system call. `pH_process_syscall` implements the monitoring, response, and training logic. pH is controlled through its own system call, `sys_pH`, which allows the superuser (root) to take the following actions:

- Start, stop monitoring processes.
- Set system parameters (see Section 3 for descriptions):
  - $delay\_factor$
  - $abort\_execve$
  - $mod\_minimum$
  - $normal\_minimum$
  - $normal\_ratio$
  - $tolerization\_limit$
  - $anomaly\_limit$
- Turn on/off logging of system calls to disk (expensive, used for debugging).
- Turn on/off logging novel sequences to disk.
- Status (prints out current values of system parameters to the kernel log).
- Write all profiles to disk.
- Reset <pid>: Resets the profile to be empty.
- Start normal <pid>: Copies the training array for $pid$'s executable to its testing array, and marks the profile as normal.
- Tolerize <pid>: Change the normal flag for $pid$'s profile to 0, reset its locality frame, and cancel any current delay for it.
- Sensitize <pid>: Clears the training array. This mechanism is used to prevent known true positives from being incorporated into the training data.
- Turn on/off debugging messages sent to kernel logging facility.

More specifically, we extended the Linux task structure (the kernel data structure used to represent processes and kernel-level threads) with a new structure which contains the following fields: the current window of system calls for the task, a locality frame, and a pointer to the current profile. A profile is a structure containing two byte-arrays for storing pairs (the training and testing arrays) and some additional training statistics described in Section 3.

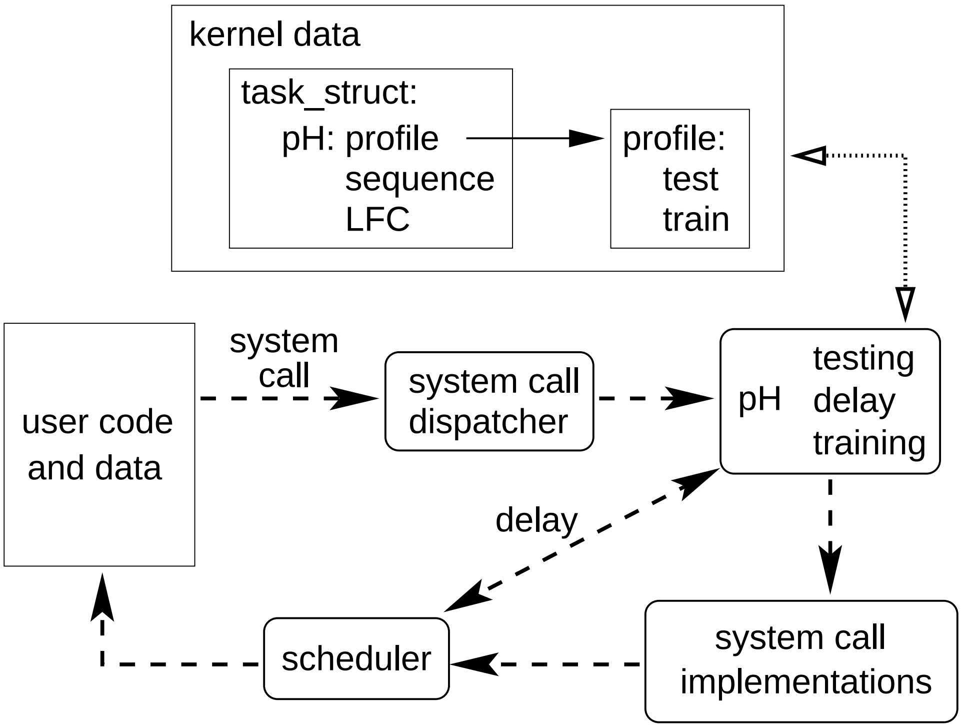


Figure 1: Basic flow of control and data in a pH-modified Linux kernel.

## 5 Experimental Results

In this section, we report on some early experiments testing out pH in a live environment. We are interested in three aspects of the system: Its effectiveness in intrusion response (can it really detect and stop an attack before the system is compromised?), performance impact (what is the overhead of the installed system?), and usability (what is it like to live with pH on your own computer?).

### 5.1 Can pH detect and stop attacks in time to prevent system compromise?

To test how pH could respond to security violations, we tested its behavior by seeing how it could detect and respond to a Secure Shell (SSH) daemon [29] backdoor, an SSH daemon buffer overflow, and a sendmail [13] attack that exploits a bug in the Linux kernel's capabilities code. These three violations all allow an attacker to obtain root privileges, using different techniques to gain access. Delays alone are significant inhibitors of these attacks; with execve aborts, pH can effectively stop all of them.

To test the SSH attacks, the `sshd` program in Debian 2.2's packaged version of Secure Shell (ssh-nonfree), version 1.2.27-6 was modified in two basic ways. First, it was made to link against the RSAREF2 library, to make it vulnerable to a buffer overflow attack script published on the BUGTRAQ mailing list [2]. Second, the source was modified using the rkssh5 trojan patch [37], and was built using the "global password" flag. This option allows an attacker to access to any account on the system using a compiled in, MD5-encoded password. In addition, use of this password disables most logging, minimizing the evidence of the intrusion.

A normal profile for this modified `sshd` binary was created by exercising the program on a personal workstation. Normal logins via root and a regular user were tested, using the password, RSA-secured rhosts, and pure RSA methods of authentication. Also, failed logins were tested, using nonexistent users and incorrect passwords. Together these produced 687 sequences, and a profile with 1725 pairs, over 47756 system calls.

Relative to this synthetic normal profile, we first tested whether pH could detect the use of the global password to gain access to the root account. With all responses disabled, the backdoor produced 5 anomalies, 3 in the child (which exec's the remote user's shell), and 2 in the

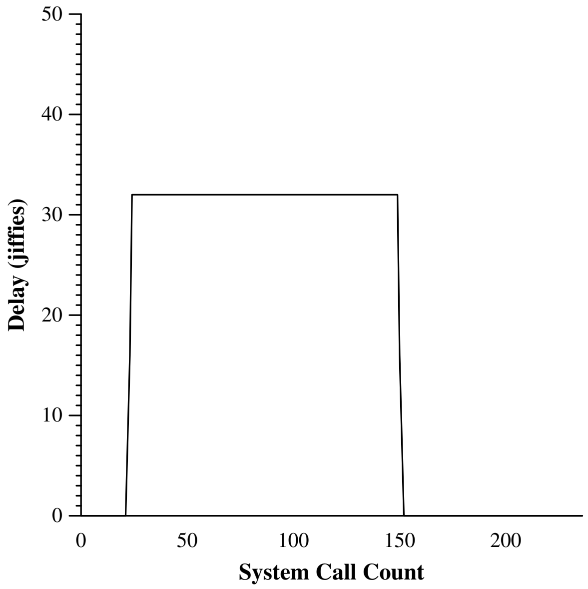


Figure 2: A graph showing the pH-induced system-call delay during the `sshd` backdoor intrusion. Note the exponential increase (from 0 to 8, 16, then 32) and decrease, with a constant delay for most calls within the locality frame. The process shown is the child process, and it terminates with a shell being exec'd. The pair window size is 6, the locality frame size is 128, and the $delay\_factor$ is set to 4. Time is measured in jiffies, which are 1/100 of a second on Linux running on i386-compatible machines.

parent (which maintains the network connection). Setting $delay\_factor$ to 4 produced the same anomaly profile, but did not prevent the remote user from logging in; however, the resulting connection was slowed down significantly, as shown in Figure 2. With $abort\_execve$ set to 1, the backdoor was closed, whether delays were enabled or not.

With all responses disabled, the buffer overflow attacked produced 4 clustered anomalies in the parent SSH process. Setting $delay\_factor$ to 4 produced the same anomalies, and allowed the attacker to obtain a root shell; however, this shell was less useful than might be supposed. Recall that pH delays every process with a non-zero LFC, and the LFC is only updated if the program has a valid normal (test) profile. As it turns out, `bash`, the standard shell on most Linux systems, is a large, complicated program that tends not to reach a stable profile. Thus, the 64 jiffy (0.64s) delay incurred by the overflowed sshd is passed on to the exec'd bash, and bash keeps this delay forever! Even if this weren't the case, because of the 128 entry locality frame, we'd see the delay for 125 system calls, giving us at least an 80s delay. Not a huge amount of time, but possibly enough to make a cracker think the attack isn't working.

With execve aborts enabled, the overflow attack was stopped, whether delays were enabled or not. The attack script does not simply fail, though; since the overflow code keeps retrying the execve call if it fails, the aborts cause an infinite loop. Each pass through the loop generates 3 anomalies, due to the failed execve; a few times through this tight loop thus causes the $tolerization\_limit$ to be exceeded, causing `sshd`'s training profile to be reset.

The Linux capability vulnerability allows a non-privileged program to prevent a privileged one from dropping its capabilities on systems running recent 2.2 kernels (2.2.14 and 2.2.15 are both vulnerable). An exploit was published on BUGTRAQ [28] which uses sendmail to take advantage of this hole. Because this is a flaw in the kernel, it can succeed even though sendmail does the right thing and tries to drop its privileges.

A normal profile for sendmail (Debian version 8.9.3-22) was first generated, based on normal usage on a personal workstation. This normal had 3443734 system calls with 1061 unique sequences, and produced a profile with 2412 system call pairs. Relative to this normal, the exploit was extremely noticeable. The exploit generates several different sendmail processes, and just one of them had 47 anomalies! Indeed, the numerous anomalies caused the $tolerization\_limit$ to be reached numerous times. Enabling execve aborts did nothing to inhibit the attack; this makes sense, since the exploit doesn't have sendmail directly run a privileged shell; instead, it creates a setuid-root shell in /tmp. However, a $delay\_factor$ of 4 effectively stopped the attack — delays were produced which lasted for at least two hours. Time delays of this magnitude would almost certainly frustrate a normal cracker; a patient one could be addressed by automatically killing any process that had been delayed for a long time period, say 30 minutes or more.

### 5.2 What is the overhead of running pH?

To determine the performance impact of our kernel modifications, we ran the HBench-OS 1.0 [11] low-level benchmark suite on an HP Pavilion 8260 (266 MHz Pentium II, 160M SDRAM, Maxtor 91020 10G UltraDMA IDE hard disk) running a pre-release version of Debian/GNU Linux 2.2. Tests were run for ten iterations on a system running in single user mode. In Tables 1 and 2, "Standard" refers to a stock Linux 2.2.14 kernel. "pH" refers to a 2.2.14 kernel with pH extensions, with monitoring enabled for all processes and with status messages and automated response turned off. All times are in microseconds.

Tables 1 and 2 show that our modifications add significantly to system call overhead. Table 1 indicates that pH adds approximately 4.7 $\mu s$ to the execution time of simple system calls that normally would take less that 2 $\mu s$ to execute. Table 2 shows that pH causes process creation to be almost twice as slow for a dynamically-linked shell. Although these tables show a significant performance hit, they are not indicative of the impact on overall system performance.

Table 3 shows how overall performance is affected for a set of tasks. Here we report the output of `time` for three different kinds of operations: kernel builds, `find / -print > /dev/null` (a basic traversal of the file system), and Quake 2 frame rates. All of these tests were run in single-user mode. The most dramatic effect is seen in the system time of the kernel build, which almost doubles due to monitoring overhead. This difference, however, only causes a 4% slowdown in the clock time. The find test shows almost a 10% slowdown, and this is for a program that is almost entirely bound by the speed of filesystem-access system calls. Interestingly, the Quake 2 frame rate tests shows virtually no slowdown. These tests illustrate what we have observed informally by using the system ourselves: If delays are

| System Call | Standard ($\mu s$) | pH ($\mu s$) |
|---|---|---|
| getpid | 1.1577 (0.00000) | 5.8898 (0.00025) |
| getrusage | 1.9145 (0.00000) | 6.6137 (0.00138) |
| gettimeofday | 1.6703 (0.00184) | 6.3779 (0.00112) |
| sigaction | 2.5609 (0.00010) | 7.2928 (0.01029) |
| write | 1.4135 (0.00187) | 6.1637 (0.00075) |

Table 1: System call latency results. All times are in microseconds. Standard deviations are listed in parentheses.

| Operation | Standard ($\mu s$) | pH ($\mu s$) |
|---|---|---|
| null | 408.80 (00.618) | 2497.90 (40.923) |
| simple | 2396.24 (11.124) | 8206.62 (11.795) |
| /bin/sh | 9385.66 (26.761) | 18223.96 (26.777) |

Table 2: Dynamic process creation latency results. Null refers to a fork of the current process. Simple is a fork of the current process plus an exec() of a hello-world program written in C. /bin/sh refers to the execution of hello-world through the libc system() interface, which uses /bin/sh to invoke hello-world. All times are in microseconds. Standard deviations are listed in parentheses.

turned off, a user can use the modified workstation without noticing any differences in system behavior, even if she decides to run a compute and I/O intensive application such as Quake 2.

### 5.3 pH in Practice

To understand the usability of the prototype, the modified kernel was installed on the authors' personal computers, configured to monitor every process on the system. As indicated above, such a configuration has a minimal performance impact in practice; however, enabling delays in this situation can cause certain problems. Privileged programs, such as `login`, `sendmail`, and `cron`, have a highly constrained behavior profile; thus, after a day or two of sampling, these programs tend to settle into a stable normal, and exhibit few anomalies. Large non-privileged programs, such as netscape and emacs, have more complicated behaviors, and thus tend not to shift into a normal monitoring mode, and so are never delayed.

Some of the more interesting programs are ones which perform simple system monitoring, such as `asclock` (a NeXTStep-style clock) and `wmapm` (a battery monitoring program). These programs execute a large number of

| Benchmark | Standard | pH |
|---|---|---|
| kernel build (s) | | |
| real | 702.47 (0.07) | 727.44 (0.29) |
| user | 669.35 (0.60) | 673.67 (0.55) |
| sys | 33.00 (0.61) | 53.60 (0.70) |
| find / -print (s) | | |
| real | 5.68 (0.58) | 6.24 (0.54) |
| user | 1.61 (0.09) | 1.59 (0.09) |
| sys | 3.27 (0.09) | 3.90 (0.17) |
| Quake 2 (fps) | | |
| demo1 | 22.89 (0.03) | 22.87 (0.05) |
| demo2 | 23.30 (0.00) | 23.30 (0.00) |

Table 3: Overall system performance. All units are seconds (s), except for the Quake 2 test, which is in frames per second (fps). Ten trials were run for each example, except 100 trials were run for `find`. Each test was run once before beginning the measurements in order to eliminate initial I/O transients. Standard deviations are listed in parentheses.

system calls, and most of the time they have repetitious behavior. However, when a user perturbs the system by changing desktops or by moving windows, the behaviors of these programs can change. In the current prototype, these programs tend to be the first to obtain normals, and the first to be slowed down. Over a few days they tend to settle down and operate normally; this transition, however, can require a number of user-supplied tolerization events. This suggests that the heuristics described in Section 3 may need to be refined. However, by temporarily setting $anomaly\_limit$ to a low value (such as 5), the number of reported anomalies can be kept to a minimum.

As monitoring programs are generally not critical applications, problems involving them can be seen as minor nuisances. A more significant set of issues arises with the behavior of one large, privileged program: the X server. The X server is responsible for initializing and controlling access to the video display, on behalf of X clients such as `xterm` and netscape. X servers are similar to monitoring programs, in that they make a large number of mostly-repetitive system calls, and so tend to acquire a normal profile quickly. User actions can also perturb the X server's behavior, causing it to be delayed. In this case, the delays can have dramatic effects, such as causing a user's entire session to be frozen or leaving the video hardware in a confused state when they occur during server initialization or shutdown. Fortunately, most of these problems can be avoided by initially starting up and shutting down the X server a few times, allowing pH to learn the critical initialization and shutdown system call patterns.

These two classes of problems suggest a weakness in our current approach. Programs which make large numbers of system calls in a short period of time tend to acquire normal profiles, even when a true sampling of behavior has not yet occurred. A natural solution is to take time into account during the normal profile decision process. Such a strategy might require a significant amount of computation, and so is probably better implemented in a userspace control daemon. It would also allow additional factors to be considered, such as size of executable, number of invocations, and perhaps program-specific heuristics. Such a daemon is planned for the future.

## 6 Related Work

Our approach to homeostatic computing is similar in spirit to Brooks' approach to mobile robot control, based on loosely coupled feedback loops, real-time interactions with dynamic environments, and no centralized representation of the outside world [9, 10]. We believe Brooks' subsumption architecture can be applied to the construction of a computer security system. pH in its current form is analogous to feedback loops that help a robot maintain balance; with the addition of a parameter-adjusting control daemon, we may be able to teach pH how to "walk."

Although research IDSs have performed anomaly detection for years [1, 27, 17, 16], most commercial systems emphasize misuse detection (i.e. pattern matching for known attacks), requiring frequent updates as new exploits are developed. Many current commercial network IDSs [3, 12, 22] are capable of automatically responding to network attacks through increased logging, firewall reconfiguration, termination of connections, and even automatic blocking of suspicious networks. Combined host and network IDSs such as ISS RealSecure [22] can also respond to threats by terminating individual processes. However, because responses that halt attacks can also cause significant service reductions, these responses must be reserved for attacks which can be easily and reliably identified through specific misuse signatures. Although useful for high-security installations, actions such as session capture and email/pager notification are simply a burden to most administrators.

Sekar, Bowen, and Segal [30] have developed a specification-based approach for intrusion detection and automated response at the system-call level. They have created a language called ASL for specifying program behavior and responses to abnormal behavior, and they have created Linux kernel extensions which allow their specifications to be enforced on-line. Their approach has the advantage of allowing subtle responses to security violations, ranging from changing system call arguments to confining a program to an alternative file system. Unfortunately, it also has the disadvantages of being labor-intensive, in that specifications must be constructed manually for each executable.

Michael Ernst and others at the University of Washington have developed techniques for dynamically determining program invariants [15]. pH also dynamically detects invariants in program behavior, although it does so at the system-call instead of the data-structure level. Perhaps Ernst's techniques could be used to create an on-line data monitoring tool which would complement the system-call monitoring of pH.

Delays are used throughout computing to achieve varying goals. Most laptop CPUs have the ability to run at a slower speed to minimize heat or maximize battery life; Transmeta's Crusoe processor [14] goes a step further by allowing the speed of the chip to vary continuously in response to system load, maximizing battery life and perceived performance. The Ethernet protocol arbitrates wire access by having transmitting computers exponentially delay their packets when collisions are detected [36]. And, at the software level, the standard `login` program on most UNIX systems delays repeated login attempts to interfere with password guessing attacks. A final example is the program `getty`, which notices if it spawns processes too frequently on a given tty device and in this event, puts itself to sleep for a few minutes.

The core of pH can be seen as an unusual type of process scheduling. In most UNIX systems [4], processes are scheduled using static priorities (provided by the administrator), dynamic priorities (based on recent CPU and I/O behavior), and the number of processes on the system. "Fair share" schedulers divide CPU time between users, not processes [18, 24]. pH's delay mechanism could be viewed as an implicit mechanism for allocating CPU time; however, instead of being fair to all processes or users, it favors processes which are behaving "normally."

Research on high-performance operating systems emphasizes extensible [5, 31] and minimal [23] kernels. These systems require novel security mechanisms to moderate the increased power given to application programs, relative to operating systems with conventional, monolithic kernels. In contrast, our work on biologically-inspired OS extensions assumes a conventional kernel, and aims to increase the stability and security of the system.

Adaptive, on-line control has been widely studied as a method for improving system performance. Whether motivated by non-stationary workloads [7], extensible operating systems [32], parallelism [25], or on-line database transaction processing [39], researchers have focused on using adaptive methods for improving system performance, not robustness. Work in using adaptive control in real-time systems [6] has focused on using adaptation to help meet timing and robustness constraints.

Finally, pH can be seen as a type of fault tolerant system [8, 33, 19], except that we focus on security violations instead of hardware or software failures.

## 7 Discussion

A major point of this paper is that it is feasible to use system-call delays to stop intrusions in real-time, without prior knowledge about what form an attack might take (unlike signature-based scanners). The three example exploits help show that pH can do this, even for very different types of attacks. However, in practice pH's effectiveness is determined by whether it can obtain stable normals for the binaries on a system. Currently, pH can do this automatically only for programs which are relatively simple and are called on a regular basis; even then, there is an ongoing risk that pH could be trained to accept intrusions as normal behavior. Research still needs to be done on more effective training heuristics that minimize the time for pH to obtain a normal profile, but also minimize the chances of pH tolerizing truly abnormal behavior. By incorporating such heuristics into a pH control daemon, we should be able to minimize the need for user or administrator intervention.

It may be necessary to implement a default timeout mechanism through pH, in which any process that is delayed beyond a certain point is automatically terminated. It may also be necessary to increase pH's repertoire to include actions such as system call parameter modifications. Additional response mechanisms may require computationally expensive analysis algorithms to be added; because abnormally-behaving processes are delayed, pH actually has the time to perform more sophisticated analysis when anomalies are detected. Our philosophy, however, is to wait until such a need arises before implementing additional mechanisms.

A second major point of the paper is to show that system-call monitoring is practical, even when every executing process on the system is monitored simultaneously. pH routinely monitors every system call executed by every process with little perceptible overhead. Thus, we believe that the current implementation of pH is efficient enough to satisfy a wide variety of users.

The current version of pH is not completely secure. pH does restrict use of the `sys_pH` system call to users who have the kill capability (which, by default is only root); however, there are no checks to ensure that a profile has not been tampered with on disk, or restrictions on user access to profiles — they are currently owned by root, but readable by anyone. An attacker could use this information to design a less-detectable attack based on the system call usage on the target machine. pH could be used to generate a denial-of-service attack by triggering abnormal (but otherwise benign) behavior in a target program. Also, it may be useful to implement mechanisms to prevent users (including root) from being able to directly modify the stored profiles. Such "hardening" of pH, though, should wait until pH's basic functionality has undergone further testing.

In the past, we have emphasized that system call profiling is a suitable technique for monitoring privileged programs. pH in its current form, however, monitors and responds to anomalies in all programs. In the future, we may decide to restrict monitoring to privileged programs; yet, with the increasing use of active content on the Internet, it may also be desirable to have pH respond to anomalies in word processors and web browsers. Some large programs such as netscape are implemented using userspace threads, causing system calls to be interleaved in apparently random patterns due to variations in thread scheduling; thus, the system call profiles of these programs may never stabilize. We believe, though, that this will be less of a problem in the future, as programs switch to using kernel threads. Because the Linux kernel uses the same data structure to represent threads and processes, pH is able to monitor kernel threads individually, avoiding interleaving effects.

## 8 Acknowledgments

The authors gratefully acknowledge the support of the National Science Foundation (grant IRI-9711199), the Office of Naval Research (grant N00014-99-1-0417), and the Intel Corporation.

Steven Hofmeyr wrote the original program for analyzing system call traces, Julie Rehmeyr rewrote the code so that it was suitable to run in the kernel, and Geoff Hunsicker developed the original login trojan, which we ported for these experiments. Margo Seltzer suggested some of the benchmarks used in the paper. Erin O'Neill pointed out to us that the immune system is better thought of as a system for maintaining homeostasis than as a defense mechanism. We are grateful to the above people and all the members of the Adaptive Computation group at UNM, especially David Ackley, for their many helpful suggestions and interesting conversations about this work.

## 9 Availability

The current version of pH may be obtained via the following web page:

http://www.cs.unm.edu/∼soma/pH/

The distribution contains a kernel patch and a few support programs. All are licensed under the terms of the GNU General Public License (GPL).

## References

[1] Debra Anderson, Thane Frivold, and Alfonso Valdes. Next-generation intrusion detection expert system (NIDES): A summary. Technical Report SRI–CSL–95–07, Computer Science Laboratory, SRI International, May 1995.

[2] Ivan Arce. SSH-1.2.27 & RSAREF2 exploit. BUGTRAQ Mailing list (bugtraq@securityfocus.com), December 14 1999. Message-ID: <3856C3EF.230F0AE@core-sdi.com>.

[3] Axent Technologies, Inc. Netprowler. http://www.axent.com, 2000.

[4] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[5] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284, Copper Mountain, CO, 1995.

[6] Thomas E. Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[7] P.R. Blevins and C.V. Ramamoorthy. Aspects of a dynamically adaptive operating system. *IEEE Transactions on Computers*, 25(7):713–725, July 1976.

[8] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under unix. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[9] Rodney A. Brooks. A robust layered control system for a mobile robot. A.I. Memo 864, Massachusetts Institute of Technology, September 1985.

[10] Rodney A. Brooks and Anita M. Flynn. Fast, cheap, and out of control: a robot invasion of the solar system. *Journal of The British Interplanetary Society*, 42:478–485, 1989.

[11] A. Brown and M. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, June 1997.

[12] Cisco Systems, Inc. Cisco secure intrusion detection system. http://www.cisco.com/warp /public/cc/cisco/mkt/security/nranger/tech /ntran_tc.htm, 1999.

[13] Sendmail Consortium. sendmail.org. http://www.sendmail.org/, 2000.

[14] Transmeta Corporation. Crusoe processor: Longrun technology. http://www.transmeta.com /crusoe/lowpower/longrun.html, January 2000.

[15] Michael D. Ernst, Adam Czeisler, William G. Griswold, , and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 7–9 2000.

[16] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*. IEEE Press, 1996.

[17] L. T. Heberlein, G. V. Dias, K. N. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Press, 1990.

[18] G.J. Henry. The fair share scheduler. *Bell Systems Technical Journal*, 63(8):1845–1857, October 1984.

[19] M. A. Hiltunen and R. D. Schlichting. Adaptive distributed and fault-tolerant systems. *Computer Systems Science and Engineering*, 11(5):275–285, September 1996.

[20] S. Hofmeyr, A. Somayaji, and S. Forrest. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[21] Steven A. Hofmeyr. *An Immunological Model of Distributed Detection and its Application to Computer Security*. PhD thesis, University of New Mexico, 1999.

[22] Internet Security Systems, Inc. RealSecure 3.0. http://www.iss.net, 1999.

[23] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malô, France, October 1997.

[24] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

[25] D.M. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.

[26] Erin O'Neill. Personal Communication, October 1998.

[27] P. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings National Information Systems Security Conference*, 1997.

[28] Wojciech Purczynski. Sendmail & procmail local root exploits on Linux kernel up to 2.2.16pre5. BUGTRAQ Mailing list (bugtraq@securityfocus.com), June 9 2000. Message-ID: <Pine.LNX.4.21.0006090852340.3475-300000@alfa.elzabsoft.pl>.

[29] SSH Communications Security. SSH secure shell. http://www.ssh.com/products/ssh/, 2000.

[30] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*. The USENIX Association, April 1999.

[31] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 1996 Symposium on Operating System Design and Implementation (OSDI II)*, 1999.

[32] Margo Seltzer and Christopher Small. Self-monitoring and self-adapting systems. In *Proceedings of the 1997 Workshop on Hot Topics on Operating Systems*, Chatham, MA, May 1997. http://www.eecs.harvard.edu/~vino/vino/papers/monitor.html.

[33] E. Shokri, H. Hecht, P. Crane, J. Dussault, and K.H. Kim. An approach for adaptive fault-tolerance in object-oriented open distributed systems. *International Journal of Software Engineering and Knowledge Engineering*, 8(3):333–346, September 1998.

[34] A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a computer immune system. In *New Security Paradigms Workshop*, New York, 1998. Association for Computing Machinery.

[35] SPI. Debian. http://www.debian.org/, 2000.

[36] Andrew S. Tanenbaum. *Computer Networks*, chapter 3, pages 145–146. Prentice Hall PTR, Englewood Cliffs, NJ, 2nd edition, 1989.

[37] timecop. Root kit SSH 5.0. http://www.ne.jp/asahi/linux/timecop/, January 2000.

[38] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, Los Alamitos, CA, 1999. IEEE Computer Society.

[39] G. Weikum, C. Hasse, A. Monkeberg, and P. Zabback. The COMFORT automatic tuning project. *Information Systems*, 19(5):381–432, July 1994.