# Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach

Xuxian Jiang[†], Aaron Walters[†], Florian Buchholz[†], Dongyan Xu[†], Yi-Min Wang[‡], Eugene H. Spafford[†]

[†] CERIAS and Department of Computer Science
Purdue University, West Lafayette, IN 47907
{ jiangx, arwalter, buchholf, dxu, spaf}@cs.purdue.edu

[‡] Microsoft Research
Redmond, WA 98052
ymwang@microsoft.com

**Abstract**

To investigate the exploitation and contamination by self-propagating Internet worms, a provenance-aware tracing mechanism is highly desirable. Provenance unawareness causes difficulties in fast and accurate identification of a worm's break-in point (namely, a remotely-accessible vulnerable service running in the infected host), and incurs significant log data inspection overhead. This paper presents the design, implementation, and evaluation of *process coloring*, an efficient provenance-aware approach to worm break-in and contamination tracing. More specifically, process coloring assigns a "color", a unique system-wide identifier, to each remotely-accessible server or process. The color will then be either inherited by spawned child processes or diffused indirectly through process actions (e.g., *read* or *write* operations).

Process coloring brings two major advantages: (1) It enables fast color-based identification of the break-in point exploited by a worm even before detailed log analysis; (2) It naturally partitions log data according to their associated colors, effectively reducing the volume of log data that need to be examined and correspondingly, log processing overhead for worm investigation. A tamper-resistant log collection method is developed based on the virtual machine introspection technique. Our experiments with a number of real-world worms demonstrate the advantages of processing coloring. For example, to reveal detailed SARS worm contamination, only $12.1\%$ of the entire log data need to be processed. Beyond the virtual machine platform of our prototype, process coloring and logging mechanisms only incur a very small additional performance penalty.

**Keywords** Intrusion Detection, Worm Infection and Investigation, Process Coloring

## 1 Introduction

Internet worms have become more stealthy and sophisticated in their infection, exploitation, and contamination. The recent absence of large-scale worm outbreaks does not indicate that Internet worms are eliminated. Quite on the contrary, there have been reports [8, 9] suggesting that worms may deliberately avoid fast massive propagation. Instead, they attempt to lurk in infected machines and surreptitiously inflict malign contaminations such as rootkit and backdoor installation [1, 34, 45]. In the combat against worms, the following tasks are critical to the understanding of a worm's exploitation details and to the recovery of an infected host from worm contaminations: **(1)** identifying the *break-in point*, namely the vulnerable, remotely accessible service via which the worm infects the victim and **(2)** determining all contaminations and damages inflicted by the worm during its residence in the victim. To perform these tasks, various intrusion analysis

tools can be used [12, 31, 35, 36]. For example, BackTracker [36] is an advanced forensic tool that traces back an intrusion starting from a "detection point" and identifies files and processes that could have affected that detection point. The tool takes the entire log file of the host as input for the back-tracking.

Log-based intrusion analysis tools face the following challenges: **(1)** Many tools [13, 36, 56] rely on an *externally-determined* detection point, from which a forensic investigation will be initiated towards the break-in point of the intrusion. However, due to a worm's possibly long "infection-to-detection" duration, it may be days or even weeks later when such a detection point is identified. It is therefore desirable that the log data carry more information and provide "leads" to initiate more timely investigations. **(2)** Current operating systems lack a *provenance-aware* mechanism to pre-classify the log data before log analysis. On the other hand, log data generated by the system may be of large volume. As reported in [36], log data as large as $1.2GB$ can be generated within one day and need to be examined for an intrusion back-track. The uncategorized bulk log data are likely to result in long duration and high overhead in worm investigation. Although human investigators can provide heuristics (such as the "filtering rules" in [36]) to reduce the log space to be examined, such heuristics may lead to inaccuracy or incompleteness in worm investigation results. **(3)** Many log-based tools do not address *tamper-resistant* log collection, which is essential in dealing with advanced worms. As shown in Section 2.3, a commonly adopted mechanism, i.e., syscall-wrapping, for collecting system call traces can be easily circumvented during an attack.

In this paper, we present the design, implementation, and evaluation of *process coloring*, an efficient provenance-aware approach to worm break-in and contamination investigation. More specifically, process coloring associates a "color", a unique system-wide identifier, to each remotely-accessible server or process - a potential worm break-in point. The color will be either *inherited* directly by any spawned child process, or *diffused* indirectly through the processes' actions (e.g., *read* or *write* operations). As a result, any process or object (e.g., a file or directory) affected by a colored process will be tainted with the same color, as recorded in the corresponding log entry. Process coloring naturally leads to the following two key advantages:

**Color-based identification of a worm's break-in point** All worm-infected processes and contaminated objects will be tainted with the same color as the original vulnerable service, which is exploited by the worm as the break-in point. By simply examining the color of any worm-related log entry or any worm-affected object, the break-in point of the corresponding worm can be immediately identified before detailed log analysis.

**Natural partition of log data** The colors of log entries provide a natural way to partition the log. To reveal the contaminations caused by a worm, it is no longer necessary to examine the entire log file. Instead, only log entries with the same color as the worm's entry point will need to be inspected. Such partition can substantially

reduce the volume of relevant log data, and thereby improve the efficiency of worm investigation.

The practicality and effectiveness of process coloring are demonstrated using a number of real-world self-propagating worms and their variants. For each of these worms, we are able to fast identify the vulnerable networked service exploited by the worm. Moreover, reduction of inspected log data is achieved in each worm experiment. For example, for a detailed SARS worm [11] break-in and contamination investigation, only $12.1\%$ of the entire log data need to be inspected. Our prototype also addresses the important requirement of tamper-resistant log data recording. Virtual machine techniques such as VMware [10], Denali [54], Xen [24], and User-Mode Linux (UML) [22] provide a better instrumentation facility than the system call hooking mechanism to safely obtain and collect internal states, including the worm exploitation and contamination information. We adopt a technique similar to Livewire [28] and develop an extension to the UML virtual machine monitor (VMM) for tamper-resistant logging.

We in this paper focus on the application of process coloring to the investigation of Internet worms. However, we also note that process coloring is a generic, extensible mechanism and some of its potentials will be presented in Section 5.1. The rest of the paper is organized as follows: Section 2 provides an overview of the process coloring scheme, whose implementation is presented in Section 3. Experimental evaluation results are presented in Section 4. Other applications and possible attacks are addressed in Section 5. Section 6 discusses related work. Finally, Section 7 concludes this paper.

# 2 Process Coloring Approach

## 2.1 Initial Coloring

Figure 1 shows a process coloring view of a networked host system running multiple servers. A unique system-wide identifier called *color* is assigned to each server process. The color assignment takes place after the server processes have started but before serving client requests. A worm breaking into the system will need to exploit a certain vulnerability of a (colored) server process. Because any action performed by the exploited process will lead to a corresponding *color diffusion* in the host (Section 2.2), the break-in and contaminations by the worm will be evidenced by the color of the affected processes and system resources and by the color of the corresponding log entries.

Each remotely-accessible service is performed by one or more active processes in the host. For example, the Samba service will start with two different processes *smbd* and *nmbd*; and both *portmap* and *rpc.statd* processes belong to the NFS/RPC service. Such processes can be assigned the same color. However, if we

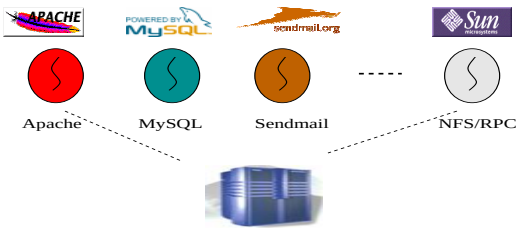



Figure 1: Process coloring view of a system running multiple servers

need to further differentiate each individual process (e.g., "which Apache process is exploited by a Slapper worm?"), different colors can be assigned to processes belonging to the same service. One benefit of such assignment is that it provides a finer granularity in log data partition. Alternatively, it is possible to define a color with two fields: a *major* field indicating the service and a *minor* field differentiating between individual working processes of the same service. For simplicity, we consider each color as having only one single field in this paper.

We note that although the process identifier (PID) uniquely identifies a process, it is *not* suitable for coloring purpose. Firstly, PIDs are generated without any awareness of break-in points. Consider a zombie process, it is not possible to tell its break-in point simply by its PID or parent's PID. Secondly, it is possible that a process dynamically injects a customized code (e.g., a whole library) into the code space of another active process. In this case, the PID is not capable of reflecting the impact of the former process on the latter. Such an attack has become popular on Windows platform (e.g., the hxdef rootkit[1]) and there exist open-source libraries (e.g., Injectso [2]) which provide similar functionality for Linux and Solaris platforms. In our design, a new field is defined in the operating system kernel to record the current colors of active processes.

## 2.2 Color Diffusion Model

After the service processes are initially colored, the colors will be diffused to other processes according to the operations performed by the processes. To reveal worm contaminations, we are especially interested in process color diffusion via system-wide shared resources, such as files, directories, and sockets. For a worm to inflict contamination (e.g., backdoor installation), it needs to go through a number of system calls. Hence the process colors are diffused to the affected system resources via the operations performed by the system calls. Table 1 shows a simplified color diffusion model with respect to several abstract operations. A worm contamination example will be described later in this section.

The color diffusion model is based on our more general *process label* framework [15], where audit information (defined as process label) is propagated and preserved in a system. We also note that process color

| *Abstract Operation* | *Color Diffusion* | *Description* | *Example Events/Actions* |
|---|---|---|---|
| $create < s_1, o >$ | $color(o) = color(s_1)$ | Subject $s_1$ creates a new object $o$ | create, mkdir, link, mknod, pipe, symlink |
| $create < s_1, s_2 >$ | $color(s_2) = color(s_1)$ | Subject $s_1$ creates a new subject $s_2$ | fork, vfork, clone, execve |
| $read < s_1, o >$ | $color(s_1) \cup = color(o)$ | Subject $s_1$ reads from object $o$ | read, readv, recv, access, stat, fstat, msgrcv |
| $read < s_1, s_2 >$ | $color(s_1) \cup = color(s_2)$ | Subject $s_1$ reads from subject $s_2$ | ptrace |
| $write < s_1, o >$ | $color(o) \cup = color(s_1)$ | Subject $s_1$ writes into object $o$ | write, writev, truncate, chmod, chown, fchown, send, sendfile |
| $write < s_1, s_2 >$ | $color(s_2) \cup = color(s_1)$ | Subject $s_1$ writes into subject $s_2$ | ptrace, kill, |
| $destroy < s_1, o >$ | - | Subject $s_1$ destroys the object $o$ | unlink, rmdir, close |
| $destroy < s_1, s_2 >$ | - | Subject $s_1$ destroys the subject $s_2$ | kill, exit |

Table 1: A simplified color diffusion model. A subject is a process while an object is a shared resource.

diffusion reflects various information flow models [14, 20, 21] in many aspects such as explicit/implicit information flows [30]. In this paper, we only consider the information flow through syscall interfaces, with the processes as subjects and intermediate resources as objects. Other means such as using CPU utilization or disk space availability to convey information are beyond the scope of this paper. In the following, we describe two types of syscall-based color diffusion:

**Direct diffusion** involves one process directly affecting the color of another process. It can happen in a number of ways: **(1)** *Process spawning:* If a process issues the *fork*, *vfork*, or *clone* system call, a new child process will usually be spawned and it will inherit the color of the parent process. **(2)** *Code injection:* A process may use code injection (e.g. via *ptrace* system call) to modify the memory space of another process to change its functionality. The color of the injected process will be updated accordingly. **(3)** *Signal processing:* A process may send a special signal (e.g., the *kill* command) to another process. If received and authorized, the signal will invoke corresponding signal handling and thus affect the execution flow of the signaled process.

**Indirect diffusion** from process $s_1$ to $s_2$ can be represented as $s_1 \Rightarrow o \Rightarrow s_2$, where $o$ is an intermediate resource (object). Various types of intermediate resources exist: some resources are dynamically created and will not exist after the process is terminated (e.g., UNIX sockets); other resources such as files can persistently exist and may later affect another process if that process acquires some input from these resources. To support indirect diffusion, the system data structure for an intermediate resource will be enhanced to record the influence of a process (i.e. its color). Later, when another process gets input from the "tainted" resource, the process will be tainted the same color [1]. Common resource types supported in current Linux systems

[1]To determine which input actually leads to an output, we show in [16] that such problem is equivalent to solving the Halting

include files, directories, network sockets (including UNIX sockets), named pipes (FIFO), and IPC (messages, semaphores, and shared memory). We also note the existence of special system-wide control resources like system timer/clock, which could be used to indirectly influence another process. However, as the information flow through the influence is usually limited (i.e., *low-bandwidth* channel) and we are not aware of any worm utilizing these special resources to affect other processes, we do not explicitly address them in this paper.

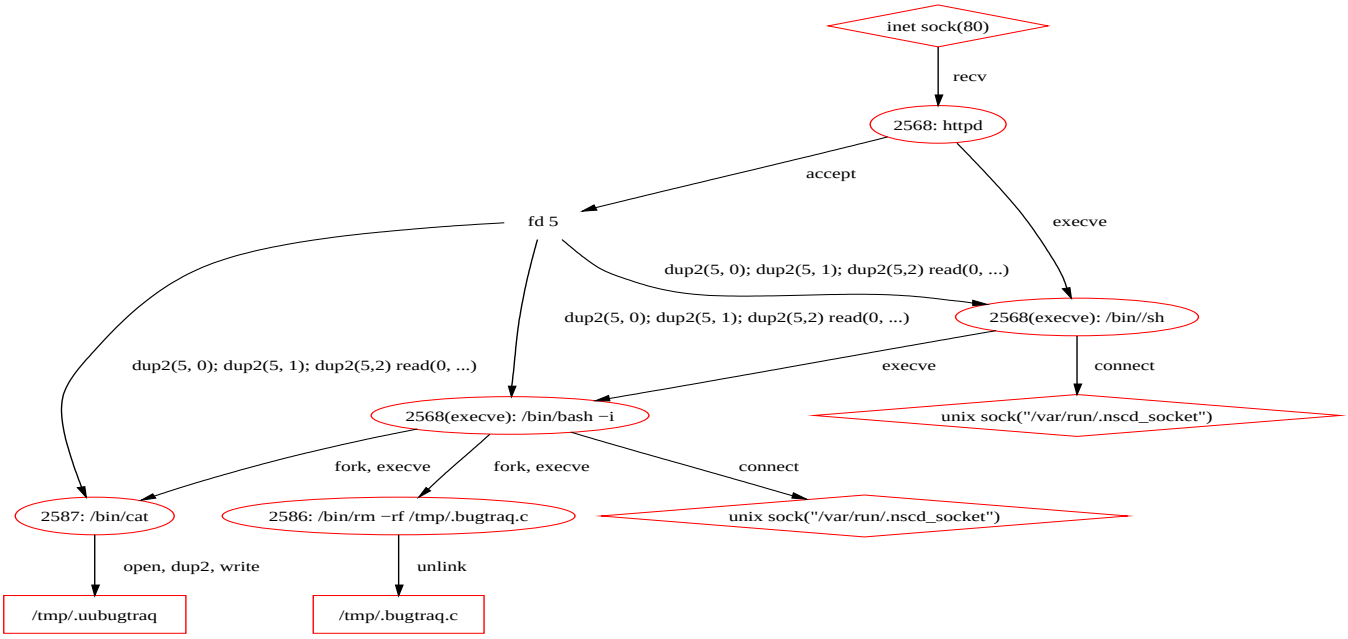
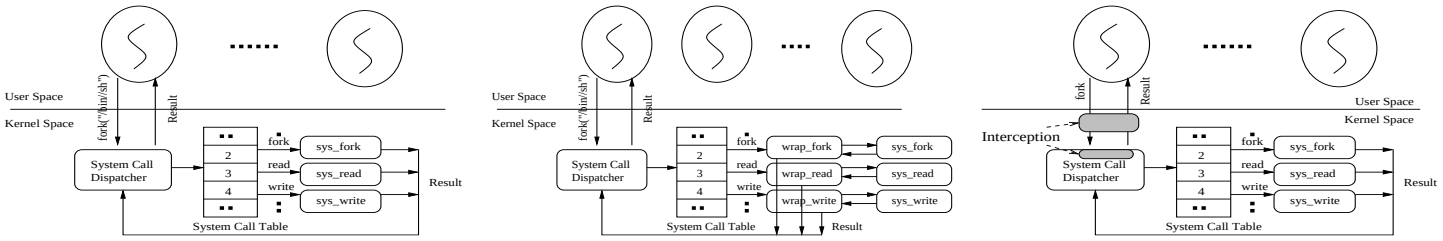



Figure 2: A coloring diffusion view showing the initial break-in of the Slapper worm

**A worm example** Figure 2 illustrates the process color diffusion during the break-in of the Slapper worm [42], which exploits a vulnerable Apache service as its break-in point. In Figure 2, an oval represents a running process, a rectangle represents a file, and a diamond represents a network socket. The number inside the oval is the PID while the following string is the name of the process. Initially, all Apache "httpd" processes are colored "RED". Right after the successful exploitation, the exploited "httpd" process (PID: 2568, color: RED) executes (*sys_execve* syscall) the program "/bin//sh" (2568, RED), which then executes (*sys_execve* syscall) the program "/bin/bash -i" (2568, RED). The "/bin/bash -i" process further spawns (by *sys_fork*) two child processes: process "/bin/rm -rf /tmp/.bugtraq.c" (2586, RED) and process "/bin/cat" (2587, RED) - their colors are inherited from their parent process via direct diffusion. Later on, the WRITE operation (*sys_write*) of process "/bin/cat" (2587, RED) updates the file (/tmp/.uubugtraq), which is thus tainted "RED". As we will show in Section 4, this file will be used to generate (*sys_read* syscall) the worm process to infect other vulnerable hosts. Via indirect diffusion, the worm process will also be colored "RED".

problem [50, 51]. To be conservative, we consider that once a process reads from a tainted source, it will also be tainted.




(a) Original system calls (b) Wrapped system calls (c) Intercepted system calls

Figure 3: Various hooking points intercepting system calls

## 2.3 Log Information Collection

Process coloring employs system call (syscall) interception to generate log entries and tag them with process colors. As demonstrated in [6, 7, 29, 36, 39, 44], syscall interception is effective in revealing and understanding intrusion steps and actions. However, a simple syscall-based hooking mechanism may be vulnerable to the *re-hooking* attack, where attackers can easily avoid or even subvert [23] the log collection function. Figure 3 compares various hooking points for syscall intercepting. Figure 3(a) shows the original implementation in the current Linux kernel. Figure 3(b) demonstrates the popular *syscall wrapping* technique to intercept system calls. Syscall wrapping modifies the system call table and redirects the corresponding calls to its own implementation. Unfortunately, if the system call table is later modified, previous interception and redirection will be invalid. This type of syscall interception is used in [6, 31, 36], which are therefore vulnerable to this re-hooking attack. Figure 3(c) shows a more advanced technique, which intercepts system calls before or while invoking the system call dispatcher. Systrace [44] implements this type of interception by modifying the system call dispatcher and thus achieves better tamper-resistance. However, it is still possible [31] for an advanced intruder to avoid the interception if the corresponding syscall interrupt handler (e.g., "int $0x80$" in Linux) is hooked in the first place.

Our design is based on the virtual machine introspection technique [28]. Though similar to Figure 3(c), the interception happens *not* in the syscall dispatcher, but *on the virtual machine virtualization path*. As such, the interceptor is an integral part of the underlying virtual machine implementation (Section 3) achieving stronger tamper-resistance. Information about each intercepted system call (e.g. current process, syscall number, parameters, return value, and return address) forms a log entry, which is tagged with the color of the current process.

# 3 Implementation

In this section, we present key aspects of process coloring implementation. Our prototype leverages User-Mode Linux (UML), an open-source VM implementation where the guest OS runs directly in the unmodified user space of the host OS, and only considers the *ext2* file system[2]. In order to support process coloring, a number of key data structures (e.g., $task\_struct$, $ext2\_inode\_info$) are modified to accommodate the color information.

## 3.1 Process Color Setting

In our prototype, a new field *color* is added to the process control block (PCB) structure, i.e., $task\_struct$, in Linux kernel. To facilitate the setting and retrieval of the *color* field, two additional system calls (*sys_setcolor* and *sys_getcolor*) are implemented. There exists a possibility that these two new syscalls might be abused to undermine process coloring. Suppose their syscall interfaces are exposed, it would be easy for worm authors to add additional code to corrupt the color assignment. Though a strong authentication scheme may be used to restrict the usage of these two syscalls, it is not desirable as it essentially achieves security by obscurity. Our solution to this problem is to create and maintain a separate color mapping table within the syscall interceptor, which allows process color setting only within a certain time period after a service starts.

## 3.2 Color Diffusion

**Direct diffusion** If a new process is created by the *fork/vfork/clone* system call, it will inherit the color of its parent process. When a process is being manipulated via the *ptrace* system call, the diffusion of color will depend on the system call parameter. If the call has parameter *PTRACE_PEEKTEXT*, *PTRACE_PEEKDATA*, or *PTRACE_PEEKUSER*, the color(s) of the ptraced process will be diffused to the ptracing process. Conversely, if the call has parameter *PTRACE_POKETEXT*, *PTRACE_POKEDATA*, or *PTRACE_POKEUSER*, the color(s) of the ptracing process will be diffused to the ptraced process. For signal processing, the color(s) of the signaling process will be diffused to the signaled process. Finally, there are system calls (*sys_waitpid* and *sys_wait4*) that will lead to color diffusion from the child process to the parent process.

**Indirect diffusion** Indirection diffusion involves an intermediate resource (object). In principle, it is feasible that the system data structure for the corresponding resource be extended to record the color information. Among all possible intermediate resources, files and directories are the two most exploited by worms. Since

[2]We are currently implementing process coloring on another VM platform Xen [24] and we expect even better performance than our UML-based prototype due to Xen's para-virtualization approach.

they are persistent resources, their colors also need to be persistently recorded. Intuitively, we can extend the corresponding *inode* data structure to accommodate the color attribute. However, adding a color field may essentially change the implementation of reading/writing files from/to a hard disk or even corrupt the underlying file system. After carefully examining all fields in current inode data structure, i.e., $ext2\_inode\_info$, we find that the field *i_file_acl* is intended to record the corresponding access control flags (ACL) but is *not* used in the *ext2* file system. In our current prototype, this field is leveraged to save the color value (represented as bitmap) of the corresponding file or directory. Note that there is another possible field, i.e., *i_dir_acl*, which is intended to record the access control flags for the corresponding directory. However, this field has already been borrowed to serve as an additional $32$-bit field for a $64$-bit file size representation for files larger than $4GB$. For non-persistent resources (e.g., IPC and network sockets), our current prototype only supports sockets, shared memory, and pipes. However, for other non-persistent resources, adding a new field is not too challenging.

### 3.3 Log Collection

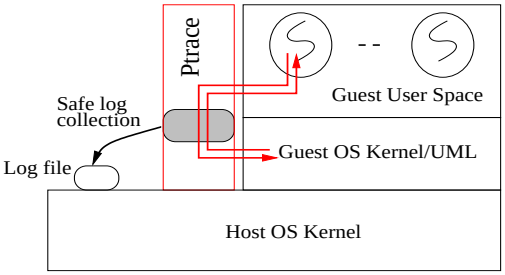



Figure 4: Tamper-resistant log collection by positioning the interceptor on the system call virtualization path

The log collection mechanism is based on the underlying virtual machine implementation, i.e. UML, as shown in Figure 4. UML adopts a system call-based virtualization approach and supports VMs in the user space of the host OS. Leveraging the capability of *ptrace*, a special thread is created to intercept the system calls made by any process in the VM, and to redirect them to the guest OS kernel. The interceptor for system call log collection is located on the system call virtualization path. Therefore, it is tamper-resistant from malicious processes running inside the VM. Moreover, once the interceptor has collected a certain amount of log data (e.g., $16K$), the log data will be pushed down to the host domain. One important benefit is that the analysis on the log file within the host domain will not interrupt the normal execution of the VM. This creates the possibility of external *runtime* system monitoring based on *colored* log data. Applications benefiting from this opportunity will be discussed in Section 5.1.

# 4 Evaluation

## 4.1 Evaluation of Run-Time Overhead

To measure the overhead introduced by process coloring, we perform a number of experiments using McVoy's LMbench [40], a suite of benchmarks targeting various subsystems of UNIX platforms. The experiments are conducted using a Dell PowerEdge 2650 server running Linux 2.4.18 with a 2.6GHz Intel Xeon processor and 2GB RAM. Three sets of experiments are performed: running LMbench on the original Linux kernel (Linux), on the unmodified UML kernel (UML), and on the modified UML kernel with process coloring capability (COLOR). The results are shown in Table 2.

| Configuration | null cal | open close | signal handler | fork | exec |
|---|---|---|---|---|---|
| Linux | 0.47 | 2.11 | 2.47 | 117 | 363 |
| UML | 11.0 | 146 | 28.5 | 4707 | 8016 |
| COLOR | 11.0 | 147 | 29.0 | 4910 | 8221 |

(a) Process-related times in $\mu s$

| Configuration | 2p/0K | 2p/16K | 2p/64K | 16p/16K | 16p/64K |
|---|---|---|---|---|---|
| Linux | 0.81 | 1.17 | 1.19 | 3.48 | 22.2 |
| UML | 9.11 | 8.75 | 9.67 | 16.7 | 46.7 |
| COLOR | 10.9 | 11.5 | 10.7 | 19.1 | 47.2 |

(b) Context switching times in $\mu s$

| Configuration | create (10K) | delete (10K) | mmap | page fault | select (100fd) |
|---|---|---|---|---|---|
| Linux | 58.8 | 10.5 | 141.0 | 1.35 | 3.197 |
| UML | 226.2 | 90.2 | 772.0 | 15.0 | 21.9 |
| COLOR | 228.6 | 90.2 | 792.0 | 15.1 | 21.9 |

(c) File and VM system latencies in $\mu s$

Table 2: LMBench results showing low additional process coloring overhead

Table 2(a) shows process operation overhead. Table 2(b) shows context switch times under varying number of processes and working set sizes. File system and virtual memory latency results are shown in Table 2(c). The results show that UML suffers a significant performance penalty due to its user-level implementation. However, process coloring only incurs a very small extra performance degradation beyond the original UML. The reason lies in the interceptor placement. By positioning the interceptor *within* the system call virtualization path, our prototype is able to avoid an additional context switch per system call, which is needed in other syscall interception schemes [39]. Also, the log data push-down is not performed upon every invocation of system call. Instead, an internal cache ($16K$) is maintained to amortize the overall disk write operations. Finally, we note that process coloring is not dependent on a specific VM platform. Moreover, we

expect that the performance penalty caused by virtualization (not by the design of process coloring) be significantly reduced with more efficient VM platforms (e.g., Xen [24] with para-virtualization) and the upcoming architecture support for VMs (e.g., Intel's Vanderpool technology [27]).

### 4.2 Experiments with Real-World Worms

We evaluate the effectiveness of process coloring using a number of real-world Internet worms: Adore [3], Ramen [4], Lion [5], Slapper [42], SARS [11], and their variants. Each worm experiment is conducted in a virtual distributed worm playground called *vGround* [32], which is a realistic, confined, and scaled-down Internet environment consisting of network entities and end hosts realized as VMs with process coloring capability. vGround makes it easy and efficient to create VMs running real-world services as well as VMs running as service clients. vGround enables safe worm experiments by confining all traffic within the vGround. It also facilitates experiments with a multi-vector worm (e.g., Ramen worm [4]), which infects different hosts (VMs) via different break-in points [3].

| | *Lion Worm* | *Slapper Worm* | *SARS Worm* |
|---|---|---|---|
| Exploited Service (CVE references) | BIND (bind-8.2.2_P5-9) (CVE-2001-0010) | Apache (apache-1.3.19-5) (CAN-2002-0656) | Samba (samba-2.2.5-10) (CAN-2003-0201) |
| Time period being analyzed | 24 hours | 24 hours | 24 hours |
| Number of log entries | 129,386 | 293,759 | 166,646 |
| Size of log data | 8.0M | 18.5MB | 10.7MB |
| Number of worm-relevant log entries | 66,504 | 195,884 | 19,494 |
| Size of worm-relevant log data | 3.9MB | 12.2MB | 1.3MB |
| Number of files "touched" by the worm | 120,342 | 62 | 200 |
| Percentage of worm-relevant logs | 48.7% | 65.9% | 12.1% |

Table 3: Statistics of process coloring log data in three worm experiments

Due to space constraint, we only present experiments with Lion, Slapper, and SARS worms. Table 3 shows key statistics of their respective log data. Each log file contains log entries collected during a 24-hour period, including both worm-related and normal service access entries. During each experiment, process coloring demonstrates its key benefits: **(1)** We are able to identify the worms' break-in points before performing detailed log analysis. The break-in points are the BIND server (bind-8.2.2_P5-9) for Lion worm, the Apache server (apache-1.3.19-5 with openssl-0.9.6b-8 package) for Slapper worm, and the Samba server (samba-2.2.5-10) for SARS worm. **(2)** The log data that need to be inspected for detailed worm investigation is only $48.7\%$ (Lion worm), $65.9\%$ (Slapper worm), and $12.1\%$ (SARS worm) of the total logged events, respectively. We note that, since log entries are naturally partitioned by their colors, increasing background service accesses

[3] For example, Ramen worm has three possible break-in points: LPRng (CVE-2000-0917), rpc.statd (CVE-2000-0666), and wu-ftp (CVE-2000-0573) - the last one cannot lead to a successful exploitation as our vGround experiment shows.

(i.e. accesses to unrelated services) in the experiments will further *reduce* the percentage of worm-related log. **(3)** Since the worm break-in point (vulnerable service) is identified *before* log analysis, it is possible to further filter the log entries that record normal accesses to the vulnerable service, which have *known and different* footprint from that of a worm infection.

### 4.2.1 Lion Worm Contamination Investigation

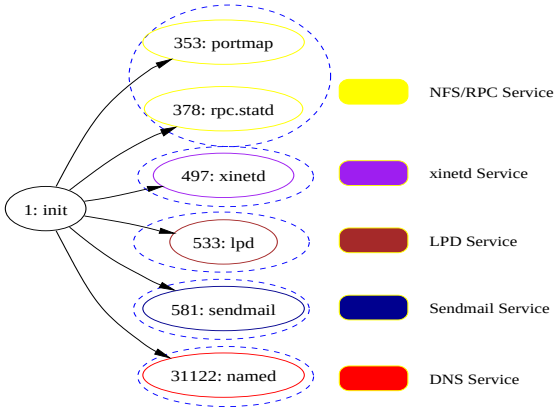



Figure 5: A process coloring view of a vulnerable system *BEFORE* Lion infection

Figure 5 shows a process coloring view of an *uninfected* system running a BIND server vulnerable to the Lion worm. There are also a number of other services hosted at the same system: NFS/RPC service (*portmap* and *rpc.statd*), printer service (*lpd*), and mail service (*sendmail*). A different color is assigned to each service. Process *named* has the color "RED" [4]. The Lion worm is unleashed from a different VM in the vGround[5]. After the experiment, we obtain a log file whose entries are conveniently partitioned by their colors. Among the "RED" entries whose provenance is the *named* process, we observe an abnormal event that a *shell* process was spawned. This is *one* of the contaminations inflicted by the Lion worm. To further reduce the inspected log volume, entries generated by normal accesses to the BIND server from other legitimate VM clients in the vGround are filtered. We then use the remaining "RED" log entries to derive a Lion worm contamination graph as shown in Figure 6[6].

We confirm that Figure 6 reveals *all* Lion worm contaminations by comparing our results with a detailed Lion worm report [5]. The leftmost oval is the vulnerable *named* daemon (PID: 31122). After a successful exploitation of the *named* process, a worm replica is downloaded (Circle 2 in Figure 6). The worm then overwrites all HTML files named *index.html* in the system with a self-carried HTML file for web defacement (Circle 3). Interestingly, we observe from the log that the worm attempts to execute the file replacement *twice*

[4]Due to the nature of process coloring, we would suggest color printing of the manuscript for review convenience.
[5]This "seed" worm is instrumented to target the vulnerable VM for infection. However, the worm copy transferred is unmodified.
[6]To view details of the worm contamination graph, we would suggest using the *zoom-in* feature of the Adobe Acrobat Reader®.

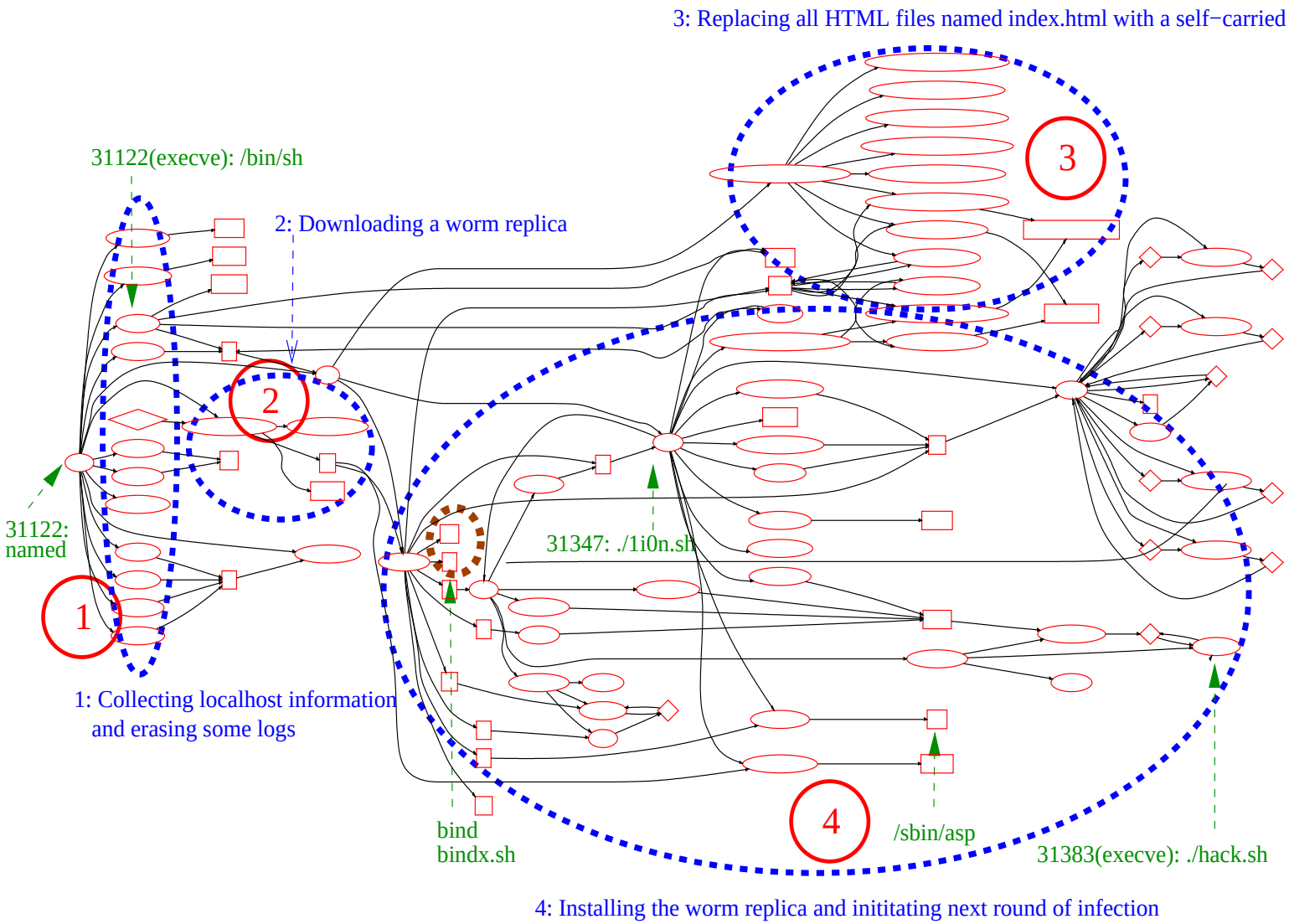



Figure 6: Lion worm contaminations reconstructed from "RED" log entries

- a detail *not* reported in [5]. The first attempt to replace files is within the shell code (PID: 31181) after executing the malicious buffer overrun code (Circle 2 and Circle 3). The second attempt happens when the driving script *./1i0n.sh* (PID: 31347) is executed (Circle 4). The worm then tries to initiate the next round of infection (Circle 4). In the thick dotted circle inside Circle 4, we find two "RED" *dangling* files *bind* and *bindx.sh*, which are introduced by the worm but never accessed by any worm-related process. Such anomaly deserves a further investigation (Section 5.1). A forensic analysis of the VM reveals that these two files contain the exploitation code for the BIND vulnerability. As there is only one VM running the vulnerable BIND service in the vGround, the worm cannot find another host to infect and the file *bindname.log* storing IP addresses of possible victims is empty. As a result, the exploitation code is never launched.

### 4.2.2 Slapper Worm Contamination Investigation

The Slapper worm experiment is conducted in a different vGround. We initially assign colors to service processes in an uninfected VM. Especially, the vulnerable Apache service is assigned "RED". Through direct diffusion, all spawned httpd worker processes are also colored "RED". A process coloring view of the system *before* the Slapper infection is shown in Figure 7. The experiment involves accesses to the other services as well as normal web accesses requesting a 2890-byte *index.html* file.

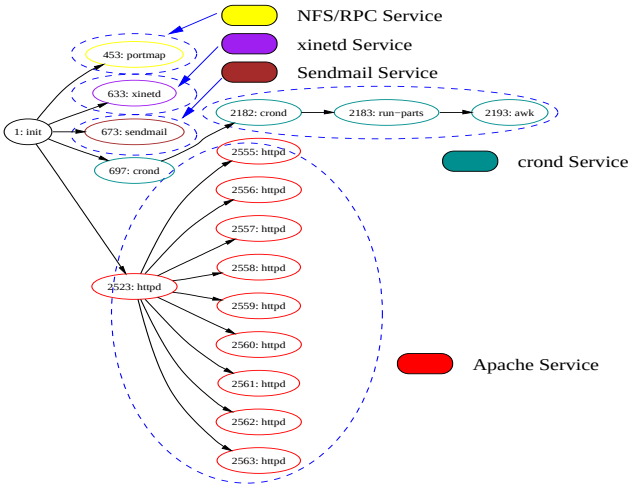



Figure 7: A process coloring view of a Slapper-vulnerable system *BEFORE* infection

After the experiment, an examination on the log file shows a flurry of "RED" log entries ($> 10000$) within a very short period (1 minute) - an anomaly indicating a possible infection. As the "RED" color is associated with the Apache web server, we select all "RED" log entries, which constitute $65.9\%$ of the entire log file. A quick review of these log entries shows that the Slapper worm infection has a large and distinct footprint in the infected host. During the transmission of a Slapper worm, an *uuencoded* source file is sent from the

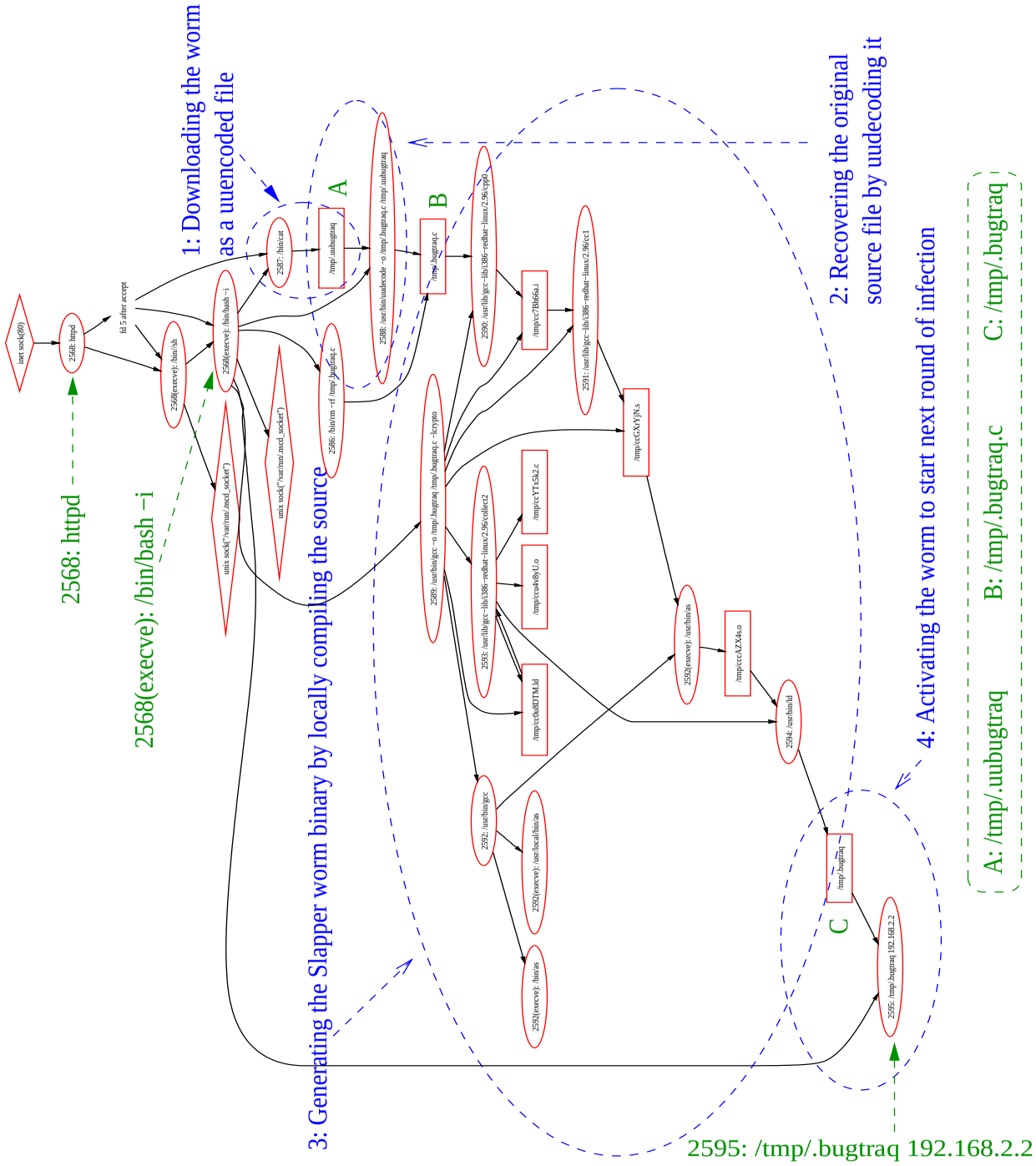



Figure 8: Slapper worm contaminations reconstructed from “RED” log entries

infecter to the victim. More specifically, the sender issues a *sendch* call for *each byte* of the uuencoded fi le. Correspondingly, the receiver uses a *sys_read* for each byte received (total 94320 calls). Moreover, each encoded byte is then written (the *cat* command) to a local file named */tmp/.uubugtraq*, leading to another 94320 *sys_write* system calls. In sharp contrast, each normal web access only generates 15 log entries, recording the known normal sequence of Apache server actions. Therefore, we remove these (“RED”) entries before

```
RED: 2523["httpd"]: 2_fork(void) = 2567 (rule 2)
RED: 2567["httpd"]: 214_setgid(48) = 0 (rule 16)
RED: 2567["httpd"]: 5_open("/etc/group", 0, 438) = 5 (rule 3)
...
RED: 2567["httpd"]: 5_open("/var/nis/NIS_COL...", 0, 438) = −2 (rule 3)
RED: 2567["httpd"]: 206_setgroups(1, 081eb4c0) = 0 (rule 49)
RED: 2567["httpd"]: 213_setuid(48) = 0 (rule 15)
...
BROWN: 673["sendmail"]: 5_open("/proc/loadavg", 0, 438) = 5 (rule 3)
BROWN: 673["sendmail"]: 192_mmap2(0, 4096, 3, 34, 4294967295, 0) = 1073868800 (rule 44)
BROWN: 673["sendmail"]: 3_read(5, "0.26 0.10 0.03 2...", 4096) = 25 (rule 4)
BROWN: 673["sendmail"]: 6_close(5) = 0 (rule 6)
BROWN: 673["sendmail"]: 91_munmap(1073868800, 4096) = 0 (rule 34)
...
RED: 2567["httpd"]: 102_accept(16, sockaddr{2, cac91f3a}, cac91f38) = 5 (rule 55)
RED: 2567["httpd"]: 3_read(5, "\1281\1\0\2\0\24...", 11) = 11 (rule 4)
RED: 2567["httpd"]: 3_read(5, "\7\0À\5\0\128\3\...", 40) = 40 (rule 4)
RED: 2567["httpd"]: 4_write(5, "\132@\4\0\1\0\2\...", 1090) = 1090 (rule 5)
RED: 2567["httpd"]: 3_read(5, "\128Ê", 2) = 2 (rule 4)
RED: 2567["httpd"]: 3_read(5, "\2\1\0\128\0\0\0...", 202) = 202 (rule 4)
RED: 2567["httpd"]: 4_write(5, "\128!\132ýFÞ\7B| ...", 35) = 35 (rule 5)
RED: 2567["httpd"]: 3_read(5, "\128!", 2) = 2 (rule 4)
RED: 2567["httpd"]: 3_read(5, "\0RØÔþn-A,÷?(\1\...", 33) = 33 (rule 4)
RED: 2567["httpd"]: 4_write(5, "\128\129ôh,\132«...", 131) = 131 (rule 5)
RED: 2567["httpd"]: 3_read(5, "(nil", 32769) = 0 (rule 4)
RED: 2567["httpd"]: 6_close(5) = 0 (rule 6)
```

Figure 9: Log excerpt showing the first exploitation of the Slapper worm attempting to get the over-writable heap address in the vulnerable Apache server. BROWN log entries are not related.

constructing the Slapper worm contamination graph (Figure 8)[7].

By comparing our results with a detailed Slapper worm analysis [42], we confirm that Figure 8 reveals *all* contaminations by the Slapper worm. We first observe that the worm exploits an httpd worker process (PID:2568) to gain system access. After that, an uuencoded version of the worm source code is downloaded (Circle 1 in Figure 8) and *uudecoded* (Circle 2) to reconstruct the original code, which is then compiled (Circle 3) to generate the worm binary. The binary is executed (Circle 4) to attempt to infect other hosts. The collected log data further reveal that the exploitation of the Slapper worm is rather sophisticated. Before the httpd worker process (PID: 2568) is exploited, 23 TCP connections have already been established with different http worker processes between the infecter and the victim. Interestingly, 21 connections among them have *no* payload; one connection is an invalid HTTP request, which turns out to be a request to obtain the Apache server version; and the last connection has a short interaction as shown in the log excerpt in Figure 9. From [42], we know that one of the 21 plain connections is used to validate the reachability of the Apache server, while the other 20 connections are made for depleting the Apache server pool to make sure that the two subsequent exploitations will have the same heap layout. The first exploitation aims at reliably deriving the over-writable heap address in the vulnerable Apache server. This heap address is then reused in the second exploitation. All these connections and interactions are recorded by "RED" log entries.

[7]We note that a general intrusion may mimic the normal sequence of service access actions [52]. However, it is more difficult for self-propagating worms to do so because their *outgoing propagation* behavior is semantically different from a normal service access.

### 4.2.3 SARS Worm Contamination Investigation

The SARS worm is a multi-platform worm, which is able to propagate across all major distributions of Linux platforms (Redhat, Debian, SuSE, Mandrake, and Gentoo) and BSD platforms (FreeBSD, OpenBSD, and NetBSD). As our current prototype is based on UML virtual machines, our experiment is conducted in a Linux-based vGround. The vulnerable Samba service is assigned "RED".

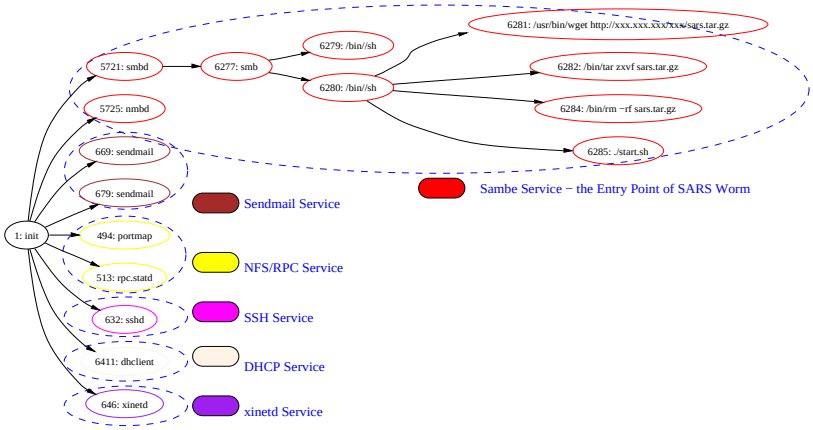



Figure 10: A process coloring view of a Redhat 8.0 system running multiple servers right *after* it is infected by the SARS worm

After the experiment, only 12.1% of the entire log data are "RED", because of the large number of log entries generated by other background services (e.g. *sendmail*, *sshd*, and *dhclient*) running in the Redhat 8.0 system. Derived from the "RED" log entries, Figure 10 shows the Redhat 8.0-based system right *after* the infection of the SARS worm. Process *smbd* (PID: 5721) and process *nmbd* (PID: 5725) have the same color ("RED") as both of them belong to the Samba service. From the figure, it seems that the exploitation code contains some redundancy as two "/bin//sh" processes are executed and one just quits immediately after its creation. Note that these two shell processes are spawned when the buffer overrun code is executed.

Continuing from process *start.sh* (PID: 6285, shown in Figure 10), Figure 11 further reveals the contaminations inflicted by the SARS worm. For readability, certain edges and nodes describing intermediate files are omitted. From the figure, we observe that the SARS worm contains a very primitive user-level *rootkit* (Circle 4 and Circle 5 in Figure 11), whose purpose is to hide the existence of worm-related files, directories, active processes, and network connections. Also, the SARS worm plants a number of backdoors such as a web server and an ICMP-based backdoor, which allow an attacker to access the infected host later. System-wide information such as host IP address, and configuration files including */etc/hosts* and */etc/passwd* is collected by the worm and sent to a hard-coded mail account (Circle 6). The integration of advanced payloads, such as the rootkit in the SARS worm, indicates a recent trend in the underground evolution of more stealthy

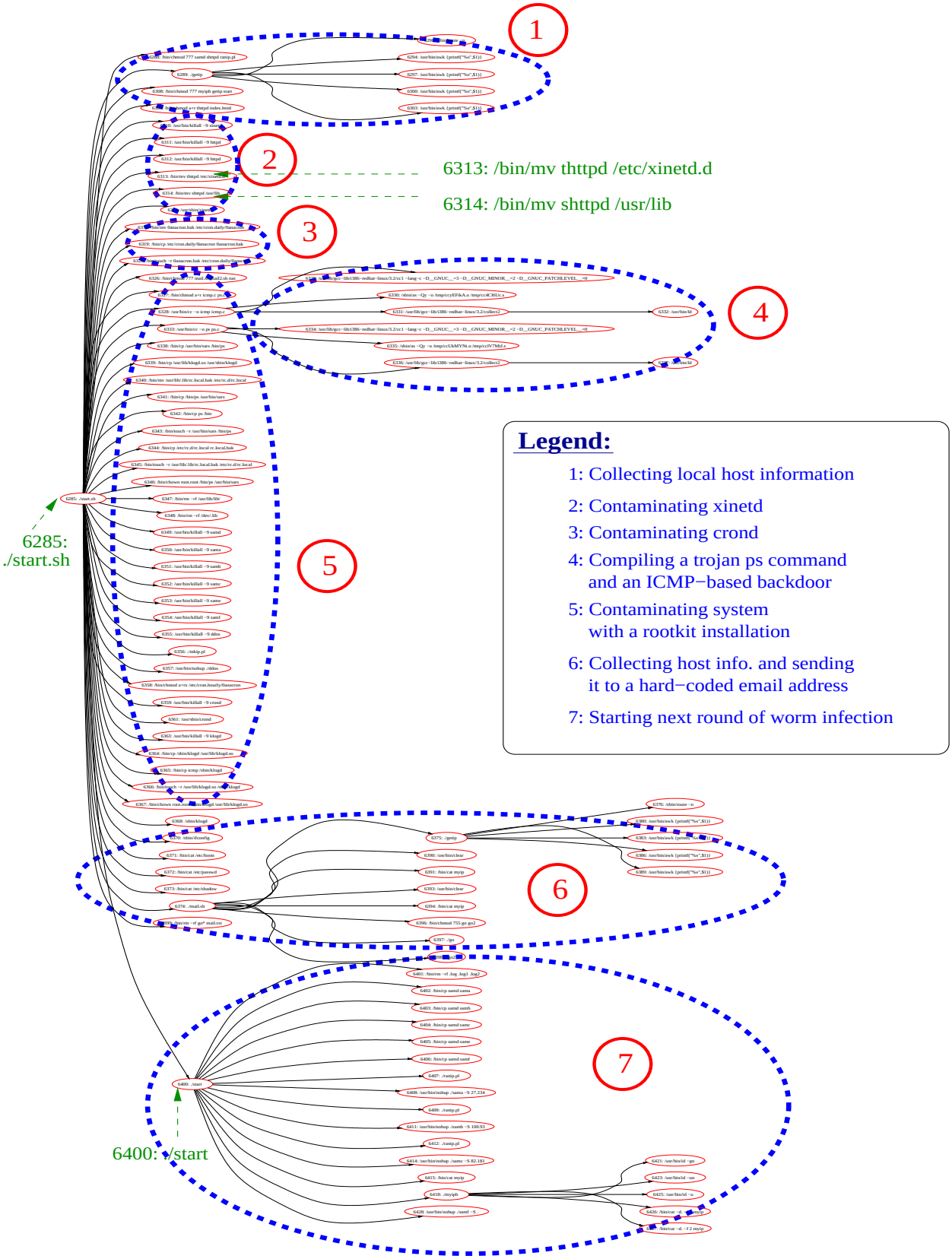



Figure 11: SARS worm contaminations reconstructed from “RED” log entries

self-propagating worms.

# 5 Other Applications and Possible Attacks

## 5.1 Other Applications

**Malware investigation** Process coloring can be naturally extended to support general malware investigation. The goal is to understand the possible malicious actions and their impact on the infected system, which can further guide the recovery from the malware's contaminations. In particular, process coloring is highly effective in exposing the following two interesting anomaly points:

- *Color mixing:* Color mixing refers to the situation where a *different* color is *diffused* to an already-colored process. Based on the rationale of color diffusion, coloring mixing indicates that the process is influenced by another process with a different color. Considering the initial assignment of different colors to mutually *unrelated* service processes, such cross-service influence is mostly likely an anomaly and warrants further investigation. For example, in one of our experiments, we run BIND and Apache services in one VM and let the Lion worm infect the VM via the BIND vulnerability. The Lion worm then contaminates the system by replacing *index.html* files with its own. We observe that log entries recording subsequent web accesses bear the colors of *both* BIND and Apache.

- *Dangling file:* A dangling file is created by a malware infection, but is not accessed during the same infection session. For example, if we re-examine Figures 6 and 11, some dangling files belong to rootkits/backdoors installed by worms: */sbin/asp* by the Lion worm (Figure 6) and */etc/xinetd.d/thttpd* and */usr/lib/shttpd* by the SARS worm (Figure 11). Though these rootkits are usually installed by stealthy worms/malware to hide their presence, identification of dangling files can actually help to reveal the presence of the rootkits.

**Run-time monitoring and log visualization** As mentioned in Section 3.3, the log push-down mechanism and color-based log partition provide a convenient means to externally monitor the running state of a networked server system, *without* interrupting the operations of the system. Process coloring can be used to identify possible anomalies revealed by log colors (e.g., color mixing, deviating log pattern for a particular color/service) during runtime and thus raise more *timely* alarms. We are currently extending our prototype with a *log visualization* tool, which makes it more intuitive for administrators to monitor system states.

## 5.2 Possible Attacks and Counter-Measures

**Jamming attack** A worm could intentionally introduce many noise log entries to hide its actual intention. For example, a worm could invoke a large number of "innocuous" or unrelated syscalls just to hide its real infection attempts. However, tactically speaking, these actions still need to be considered as a part of the worm's behavior in the infected system, even though they may not contribute to any real damage. Also, to the worm's *disadvantage*, these noise log entries deviate from the normal log pattern of a specific color and will trigger an alarm. Finally, the capability of color-based identification of a worm's entry point is still valid under this attack, though it will take a more careful analysis to uncover the obfuscated intention.

**Low-level attack** The integrity of colors associated with active processes and intermediate resources are critical to worm investigation. As the current prototype maintains the color information within the kernel of the system under inspection, it is possible that this information be manipulated through certain low-level attacks. For example, if the process color is associated with the *task_struct* PCB structure, a method called direct kernel object manipulation (DKOM) [17] can be leveraged to explicitly change the color value (e.g., by writing to the special device file */dev/kmem*). Fortunately, solutions such as CoPilot[43], Livewire[28], and Pioneer [46] have been proposed to address the issue of kernel integrity. Another possible counter-measure is to create a *shadow* structure, which is instead maintained by the virtual machine monitor (VMM) and is totally inaccessible from inside the VM. Compared with the current prototype, the shadow solution poses significantly greater challenge in deriving VM operation semantics from low-level information collected via virtual machine introspection, which may affect the accuracy and completeness of worm investigation results.

**Diffusion-cutting attack** It is possible that a worm might use a hidden channel to undermine the diffusion. For example, a worm could use an initial part of an attack to crack a weak password, which is later used in a *separate* session to gain the system access and complete the rest of worm contamination. Process coloring can track any action performed within each break-in, but it cannot automatically associate the second break-in with the first one. However, any anomaly within the second break-in will immediately expose the responsible login session, which may lead to the identification of the cracked password. Based on the log data from the first break-in, the administrator may still be able to correlate those two disjunct break-ins.

**Color saturation attack** If a worm is aware of the coloring scheme, it might attempt to acquire more colors from different services right after its break-in. As a result, the associated colors can not uniquely identify the break-in point. However, to the worm's *disadvantage*, the color saturation attack will immediately lead to an alarm of *color mixing* (Section 5.1) - an anomaly triggering further investigation. Color saturation attack does expose a weakness of our current prototype, which uses a single color field. Although our prototype is

able to accommodate multiple colors (each bit in the color field represents a different color), it is not able to differentiate between an *inherited* color and a *diffused* color. The inherited color of a process can only be inherited from its parent and will not be changed by its own or others' behavior. The diffused colors, on the other hand, reflect the color diffusions through its own or others' actions (e.g., *sys_read* and *sys_write*). With this distinction, the inherited colors can be used to partition the log data, while the diffused colors can be used to detect a color saturation attack and naturally identify all color-mixing points for further examination in affected partitions.

# 6 Related Work

The development of process coloring technique is inspired by the concept of transitive dependency tracking [26, 47, 49] originally proposed for failure recovery in fault tolerant systems. Process coloring also reflects various information flow models [14, 20, 21]. With these concepts and models as theoretical underpinnings, a spectrum of taint-based techniques have recently been proposed for different aspects of system security: Process coloring operates at the system call level to reveal worm break-in and contamination semantics; TaintCheck [41] works at the instruction level to detect overwrite attacks and generate exploit signatures; TaintBochs [18] focuses on the lifetime tracking of sensitive data (e.g., passwords) in a system. While sharing the same design philosophy, these techniques differ in their goals, design, implementation, and usage.

Process coloring can be integrated into existing log-based intrusion investigation tools [36, 38] so that they become provenance-aware. BackTracker [36] is able to automatically reconstruct the sequences of steps that occurred during an intrusion based on log data. More specifically, starting with an external detection point (e.g., a corrupted file), BackTracker identifies files and processes that could have affected this detection point and displays chains of events in a dependency graph. The follow-up work [38] of Backtracker proposes a forward tracking capability that identifies all possible damages caused by the intrusion after the back-tracking session. Both BackTracker and its forward tracking extension require the entire log data as input. With process coloring enhancement, the break-in point of a worm can first be identified by the color of the detection point, and the volume of input log data will be reduced by color-based log partition, resulting in more efficient back-tracking and forward-tracking sessions. In addition, the colors and patterns of log entries may provide alerts at runtime, leading to more timely investigations.

Process coloring can also be applied to enhance file and transaction repair/recovery systems. The Repairable File Service [56] aims to identify possible file system level corruptions caused by a root process, assuming that the administrator has already identified the root process that starts an attack or a human-involved

error. It then uses the log data to identify the files that may have been contaminated by that process. The repairable file service implements a limited version of the forward tracking capability mentioned earlier by only tracking file system-level corruptions. Meanwhile, there has been technique in the database area [13] that is capable of recording contaminations at the transaction level and rolling back the damages if the transaction is later found malicious. This technique also requires external identification of malicious processes or transactions. Process coloring can enhance these techniques by tracking more sophisticated contamination behavior via color diffusion, raising anomaly alarms based on log colors and patterns, and achieving tamper-resistant log collection.

Recent advances in virtual machine technologies have created tremendous opportunities for intrusion monitoring and replay [7, 25, 28, 33], system problem diagnosis [37, 53, 55], attack recovery and avoidance [25, 48], and data life-time tracking [18, 19]. For example, ReVirt [25] is able to replay a system's execution at the instruction level. Time-traveling virtual machines such as [37, 53, 55] provide a highly effective means of re-examining and troubleshooting system execution or configuration. Process coloring complements these efforts by leveraging virtual machine technologies for worm break-in and contamination investigation. In addition, process coloring, as an advanced logging mechanism, can be integrated into other VM-based networked systems to add provenance-awareness to these systems.

# 7 Conclusion

We have presented the design, implementation, and evaluation of process coloring, a novel systematic approach to provenance-aware tracing of worm break-in and contaminations. By associating a unique color to each remotely-accessible service and diffusing the color based on actions performed by processes in the system, process coloring achieves two key benefits: (1) color-based identification of a worm's break-in point before detailed log analysis and (2) color-based partitioning of log data. Process coloring improves log-based worm investigation tools by reducing the amount of log entries to be processed and by providing color-related "leads" for more timely investigation. Experiments with a number of real-world Internet worms demonstrate the practicality and effectiveness of process coloring.

# References

[1] hxdef. *http://hxdef.czweb.org*.

[2] Injectso: Modifying and Spying on Running Processes under Linux and Solaris. *http://www.blackhat.com/presentations/bh-europe-01/shaun-clowes/bh-europe-01-clowes.ppt*.

[3] Linux Adore Worms. *http://securityresponse.symantec.com/avcenter/venc/data/linux.adore.worm.html*.

[4] Linux Ramen Worm. *http://service1.symantec.com/sarc/sarc.nsf/html/pf/linux.ramen.worm.html*.

[5] SANS Institute: Lion worm. *http://www.sans.com/y2k/lion.htm*.

[6] Sebek. *http://www.honeynet.org/tools/sebek/*.

[7] The Honeynet Project. *http://www.honeynet.org*.

[8] The Strange Decline of Computer Worms. *http://www.theregister.co.uk/2005/03/17/f-secure_websec/print.html*.

[9] Virus Writers Get Stealthy. *http://news.zdnet.co.uk/internet/security/0,39020375,39191840,00.htm*.

[10] VMware. *http://www.vmware.com/*.

[11] SARS Worms. *http://www.xfocus.net/tools/200306/413.html*, June 2003.

[12] E. Alata, M. Dacier, Y. Deswarte, M. Kaaniche, K. Kortchinsky, V. Nicomette, V.H. Pham, and F. Pouget. CADHo: Collection and Analysis of Data from Honeypots. *Proceedings of 5th European Dependable Computing Conference*, April 2005.

[13] Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from Malicious Transactions. *IEEE Transactions on Knowledge and Data Engineering, Volume 14, Issue 5, 1167-1185*, September 2002.

[14] D. Bell and L. LaPadula. MITRE Technical Report 2547 (Secure Computer System): Volume II. *Journal of Computer Security, vol. 4, no. 2/3, pages 239-263*, 1996.

[15] F. Buchholz. Pervasive Binding of Labels to System Processes. *Ph.D. Thesis, also as CERIAS Technical Report 2005-54, Purdue University*, 2005.

[16] Florian Buchholz and Eugene H. Spafford. On the Role of File System Metadata in Digital Forensics. *Journal of Digital Investigation*, December 2004.

[17] Jamie Butler. Direct Kernel Object Manipulation (DKOM). *http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf*, 2004.

[18] J. Chow, B. Pfaff, T. Garfi nkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. *Proceedings of the USENIX 13th Security Symposium, San Diego, USA*, August 2004.

[19] J. Chow, B. Pfaff, T. Garfi nkel, and M. Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. *Proceedings of the USENIX 14th Security Symposium, San Diego, USA*, August 2005.

[20] D. R. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. *Proceedings of the 1987 IEEE Symposium on Security and Privacy, pages 184-194*, 1987.

[21] D. E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM 19, 5 (May), 236-243*, 1976.

[22] J. Dike. User Mode Linux. *http://user-mode-linux.sourceforge.net*.

[23] Maximillian Dornseif, Thorsten Holz, and Christian Klein. NoSEBrEaK - Attacking Honeynets. *Proccedings of the 5th Annual IEEE Information Assurance Workshop, Westpoint*, June 2004.

[24] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfi eld, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. *Proc. of ACM SOSP 2003*, October 2003.

[25] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[26] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message Passing Systems. *ACM Computing Survey*, 34(3), 2002.

[27] Rich Uhlig et al. Intel Virtualization Technology. *IEEE Computer, Special Issue on Virtualization Technology*, May 2005.

[28] Tal Garfi nkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection . *Internet Society's 2003 Symposium on Network and Distributed System Security (NDSS)*, February 2003.

[29] Ashvin Goel, Mike Shea, Sourabh Ahuja, Wu-Chang Feng, Wu-Chi Feng, David Maier, and Jonathan Walpole. Forensix: A Robust, High-Performance Reconstruction System. *19th Symposium on Operating Systems Principles (SOSP) (poster session)*, October 2003.

[30] J. A. Goguen and J. Meseguer. Security Policies and Security Models. *Proceedings of the 1982 IEEE Symposium on Security and Privacy, pages 11-20*, 1982.

[31] J. Grizzard, J. Levine, and Henry Owen. Re-Establishing Trust in Compromised Systems: Recovering from Rootkits that Trojan the System Call Table. *Proc. of 9th European Symposium on Research in Computer Security*, September 2004.

[32] X. Jiang, D. Xu, H. J. Wang, and E. H. Spafford. Virtual Playgrounds for Worm Behavior Investigation. *Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, September 2005.

[33] Xuxian Jiang and Dongyan Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. *Proceedings of the USENIX 13th Security Symposium, San Diego, USA*, August 2004.

[34] Kdm. Win32 Portable Userland Rootkit. *Phrack 62:article 12 of 16*, July 2004.

[35] Gene H. Kim and Eugene H. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. *In Systems Administration, Networking and Security Conference III, USENIX*, 1994.

[36] S. T. King and P. M. Chen. Backtracking Intrusions. *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*, October 2003.

[37] S. T. King, George W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. *Proceedings of the 2005 Annual USENIX Technical Conference*, April 2005.

[38] Samuel T. King, Z. Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching Intrusion Alerts Through Multi-Host Causality. *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS)*, February 2005.

[39] Z. Liang, VN Venkatakrishnan, and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. *Proceedings of the Nineteenth Annual Computer Security Applications Conference*, December 2003.

[40] L. McVoy and C. Staelin. LMbench: Portable Tools for Performance Analysis. *USENIX Annual Technical Conference*, 1996.

[41] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Proceedings of NDSS 2005*, February 2005.

[42] Frederic Perriot and Peter Szor. An Analysis of the Slapper Worm Exploit. *Symantec White Paper http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf*.

[43] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. *Proceedings of the USENIX 13th Security Symposium, San Diego, USA*, August 2004.

[44] Niels Provos. Improving Host Security with System Call Policies. *USENIX Security Symposium*, August 2003.

[45] sd. Linux on-the-fly kernel patching without LKM. *Phrack, 11(58):article 7 of 15*, December 2001.

[46] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms. *Proceedings of ACM SOSP 2005*, October 2005.

[47] A. Sistla and J. Welch. Efficient Distributed Recovery Using Message Logging. *Proceedings of ACM PODC'89*, 1989.

[48] A. Stavrou, A. D. Keromytis, J. Nieh, V. Misra, and D. Rubenstein. MOVE: An End-to-End Solution To Network Denial of Service. *Proceedings of 2005 Symposium on Network and Distributed System Security (NDSS)*, February 2005.

[49] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computer Systems*, 3(3), 1985.

[50] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungs Problem. *Proc. of London Math. Soc. Ser. 2, 42:230-265*, 1937.

[51] A. M. Turing. Correction to: On Computable Numbers, with an Application to the Entscheidungs Problem. *Proc. London Math. Soc. Ser. 2, 43:544-546*, 1938.

[52] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. *Proceedings of ACM CCS 2002*, November 2002.

[53] A. Whitaker, Richard S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. *Proc. of USENIX OSDI 2004*, December 2004.

[54] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. *Proc. of USENIX OSDI 2002*, December 2002.

[55] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Using Time Travel to Diagnose Computer Problems. *Proceedings of the 11th SIGOPS European Workshop*, September 2004.

[56] N. Zhu and T. Chiueh. Design, Implementation and Evaluation of Repairable File Service. *Proceedings of the 2003 International Conference on Dependable Systems and Networks, San Francisco, CA*, June 2003.