Tracing Worm Break-In and Contaminations via Process Coloring: A Provenance-Preserving Approach

Xuxian Jiang, *Member*, *IEEE*, Florian Buchholz, Aaron Walters, Dongyan Xu, *Member*, *IEEE*, Yi-Min Wang, *Senior Member*, *IEEE*, and Eugene H. Spafford, *Fellow*, *IEEE*

Abstract—To detect and investigate self-propagating worm attacks against networked servers, the following capabilities are desirable: 1) raising timely alerts to trigger a worm investigation, 2) determining the break-in point of a worm, i.e., the vulnerable service from which the worm infiltrates the victim, and 3) identifying all contaminations inflicted by the worm during its residence in the victim. In this paper, we argue that the worm *break-in provenance* information has not been exploited in achieving these capabilities and thus propose process coloring, a new approach that preserves worm break-in provenance information and propagates it along operating-system-level information flows. More specifically, process coloring assigns a "color," a unique systemwide identifier, to each remotely accessible server process. The color will be either inherited by spawned child processes or diffused transitively through process actions. Process coloring achieves three new capabilities: *color-based* worm warning generation, break-in point identification, and log file partitioning. The virtualization-based implementation enables more tamper-resistant log collection, storage, and real-time monitoring. Beyond the overhead introduced by virtualization, process coloring only incurs very small additional system overhead. Experiments with real-world worms demonstrate the advantages of processing coloring over non-provenance-preserving tools.

Index Terms—Networked server, Internet worm, process coloring, system monitoring, computer forensics.

1 Introduction

Internet worms have become increasingly stealthy and sophisticated in their infection and contamination behavior. The recent absence of large-scale worm outbreaks does not indicate that Internet worms are eliminated. Quite on the contrary, recent reports [6], [7] have suggested that emerging worms may deliberately avoid massive propagation. Instead, they lurk in infected machines and inflict contaminations over time, such as rootkit and backdoor installation, botnet creation, and data theft. In this paper, we focus on worm investigation in networked server environments, which involves the following tasks: 1) raising timely alerts to trigger a worm investigation, 2) determining the *break-in point* of a worm, i.e., the vulnerable service from which the worm infiltrates the victim, and 3) identifying all contaminations inflicted by the worm during its residence in the victim.

- *X. Jiang is with the Department of Computer Science, George Mason University, 4400 University Drive, Mail Stop 4A4, Fairfax, VA 22030-4444. E-mail: xjiang@gmu.edu.*
- *F. Buchholz is with the Department of Computer Science, James Madison University, MSC 4103, Harrisonburg, VA 22807. E-mail: buchhofp@jmu.edu.*
- *A. Walters, D. Xu, and E.H. Spafford are with the Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN 47907. E-mail: {arwalter, dxu, spaf}@cs.purdue.edu.*
- *Y.-M. Wang is with Microsoft Corporation, One Microsoft Way, Redmond, WA 98052. E-mail: ymwang@microsoft.com.*

Manuscript received 22 July 2006; revised 9 July 2007; accepted 24 July 2007; published online 29 Sept. 2007.
Recommended for acceptance by M. Singhal.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0201-0706.
Digital Object Identifier no. 10.1109/TPDS.2007.70765.

To perform these tasks, various log-based intrusion investigation tools have been developed [24], [25], [31], [33]. As a typical example, BackTracker [31] traces back an intrusion starting from a "detection point" and identifies files and processes that could have led to the detection point, using the entire log of the system as input. Still, current log-based intrusion investigation tools have one or more of the following limitations: 1) Many tools [24], [25], [31], [33] rely on an *externally* determined detection point from which the investigation will be initiated toward the break-in point of the intrusion. However, it may be days or even weeks before such a detection point is found. During this long "infection-to-detection" interval, the log remains a *passive* repository and does not provide "leads" to initiate more timely investigations. 2) Log data generated by a host may be of large volume. As reported in [31], log files as large as 1.2 Gbytes can be generated daily. Current tools do not preclassify log entries, and as a result, the bulk of uncategorized log data can lead to a high log processing overhead. 3) Many log-based tools do not address *tamper-resistant* log collection, whereas advanced worms tend to tamper with the log and logging facilities after break-in. For example, system call (syscall) wrapping [31], a commonly used mechanism for syscall logging, can easily be circumvented [19].

In this paper, we address the above limitations by preserving worm *break-in provenance* information and propagating it along information flows at the operating system (OS) level. We argue that the break-in provenance information has not been fully utilized in worm investigation. More specifically, we present *process coloring*, a provenance-preserving approach to worm alerts, as well

as worm break-in and contamination tracing. In this approach, a "color," a unique systemwide identifier, is associated with every potential worm break-in point, namely, every remotely accessible service process (for example, Web, mail, or DNS service process) in a server host. The color will be either *inherited* directly by any spawned child process or *diffused* indirectly through the processes' actions (for example, *read* or *write* operations) along the information flows between processes or between processes and objects (for example, files or directories). As a result, any process or object affected by a colored process will be tainted with the same color. To preserve the provenance of such influence, the corresponding log entry will also record the color. Process colors, as recorded in the log entries, reveal valuable information about possible worm break-ins and contamination actions. Process coloring will bring the following key capabilities to worm investigation:

- *Color-based determination of the worm break-in point.* All worm-affected processes and contaminated objects will bear the color of the original vulnerable service —the break-in point through which the worm has broken into the server host. By examining the color of any worm-related log entry, the break-in point can be determined or narrowed down *before* a detailed log analysis.
- *Color-based partitioning of the log file.* The log color provides a natural index to partition the log file. To reveal the contaminations caused by a worm, it is no longer necessary to examine the entire log file. Instead, only those log entries carrying the color of the worm's break-in point need to be inspected. Color-based log partitioning substantially reduces the volume of log data to be analyzed for worm contamination reconstruction.
- *Color-based worm warning.* Process coloring turns the passive log into an active generator of worm warnings based on the coloring anomalies shown in the log entries. The colors reveal the anomalous influence between processes or between processes and objects under a worm attack, which is not supposed to happen under normal circumstances. Worm warnings are generated in *real time* by monitoring the log entry colors—a new capability not provided by the non-provenance-preserving tools.

Our process coloring prototype also achieves more tamper-resistant log collection and storage. Process coloring leverages the virtualization technology, especially the virtual machine introspection (VMI) technique [23], which enables *external* (relative to the server host being monitored) log collection, storage, and monitoring. Our prototype extends the User-Mode Linux (UML) [18] virtual machine monitor (VMM) for log collection with negligible additional overhead beyond the overhead incurred by UML itself.

The effectiveness of process coloring has been demonstrated in our experiments with a number of real-world worms and their variants. For each worm experiment, we are able to receive real-time warnings that trigger a timely investigation without having to wait for an external detection point, we are able to identify the break-in point of the worm *before* a detailed log analysis, and we only have to use a subset of the log entries as input to reconstruct a full account of the worm's contaminations.

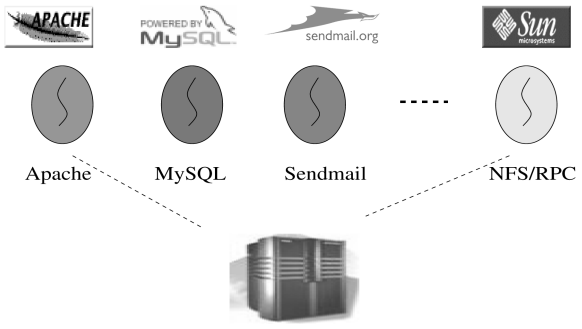


Fig. 1. Process coloring view of a networked server running multiple services.

In this paper, we focus on the application of process coloring to the investigation of worms that target networked server hosts running multiple service processes. However, process coloring is a generic extensible mechanism that may be applied to other types of malware. The rest of the paper is organized as follows: Section 2 gives an overview of the process coloring approach. Section 3 presents its implementation. Experimental evaluation results are presented in Section 4. Section 5 discusses possible attacks against process coloring. Section 6 discusses related work. Finally, Section 7 concludes this paper.

2 Process Coloring Overview

Based on the classic information flow models [10], [16], [17], [26], process coloring relies on the OS-level information flows, where the principals are processes and the objects are system objects such as files, directories, and sockets. Our new contribution lies in the preservation of worm break-in provenance information (that is, possible worm break-in points), which is defined as process colors, diffused along OS-level information flows, and recorded in log entries.

2.1 Initial Coloring

Fig. 1 shows an example of initial process coloring in a server host that consolidates multiple services. A unique systemwide identifier called *color* is assigned to each service process. A worm trying to break into the server will have to exploit a certain vulnerability of a (colored) service process. The color of the exploited process will then be *diffused* (Section 2.2) in the host, following the actions performed by the worm. As a result, the break-in and contaminations by the worm will be evidenced by the color of the affected processes and system objects and, correspondingly, by the color of the associated log entries.

A service may involve more than one process. For example, the Samba service will start with two different processes *smbd* and *nmbd*, whereas the *portmap* and *rpc.statd* processes both belong to the NFS/RPC service. These processes can be assigned the same color. However, if we need to further differentiate each individual process (for example, "which Apache process is exploited by a Slapper worm?"), multiple colors can be assigned to processes

TABLE 1
The Color Diffusion Model: A Process Is a Subject and a Shared Resource Is an Object

Abstract Operation	*Color Diffusion*	*Description*	*Example Events/Actions*
$create < p_1, o >$	$color(o) = color(p_1)$	Subject p_1 creates a new object o	create, mkdir, link, mknod, pipe, symlink
$create < p_1, p_2 >$	$color(p_2) = color(p_1)$	Subject p_1 creates a new subject p_2	fork, vfork, clone, execve
$read < p_1, o >$	$color(p_1) \cup = color(o)$	Subject p_1 reads from object o	read, readv, recv, access, stat, fstat
$read < p_1, p_2 >$	$color(p_1) \cup = color(p_2)$	Subject p_1 reads from subject p_2	ptrace
$write < p_1, o >$	$color(o) \cup = color(p_1)$	Subject p_1 writes to object o	write, writev, truncate, chmod, chown, fchown, send, sendfile
$write < p_1, p_2 >$	$color(p_2) \cup = color(p_1)$	Subject p_1 writes to subject p_2	ptrace, kill
$destroy < p_1, o >$	-	Subject p_1 destroys object o	unlink, rmdir, close
$destroy < p_1, p_2 >$	-	Subject p_1 destroys subject p_2	kill, exit

belonging to the same service or application. A benefit of such an assignment is a finer granularity of log partitioning.

2.2 Color Diffusion

After the service processes are initially colored, the colors will be diffused to other processes along OS-level information flows through processes and systemwide shared objects. More specifically, process colors are diffused via operations performed by syscalls—the OS interface that a worm uses to inflict contaminations (for example, backdoor installation). Table 1 shows a color diffusion model that accounts for an incomplete list of operations. We define two types of color diffusion:

- *Direct color diffusion* involves one process directly affecting the color of another process. This can happen in a number of ways: 1) *Process spawning.* If a process issues the *fork*, *vfork*, or *clone* syscall, the new child process will inherit the color of the parent process. 2) *Code injection.* A process may use code injection (for example, via the *ptrace* syscall) to modify the memory space of another process. 3) *Signal processing.* A process may send a special signal (for example, the *kill* command) to another process. If received and authorized, the signal will invoke the corresponding signal handler and thus affect the execution flow of the signaled process.
- *Indirect color diffusion* from process p_1 to p_2 can be represented as $p_1 \Rightarrow o \Rightarrow p_2$, where o is an intermediate resource (object). There are two types of intermediate objects: those that are dynamically created and will not exist after the relevant process is terminated (for example, Unix sockets) and those that may persistently exist (for example, files) and later affect other processes if the processes acquire certain information from them. In Linux OSs, the following types of objects are involved in process coloring: files, directories, network sockets (including Unix sockets), named pipes (FIFO), and IPC (messages, semaphores, and shared memory). To support indirect color diffusion, the OS data structures of these objects will be extended to record their colors. When a process obtains information from a colored object, the process will be tainted with that color.[1] We note that process coloring does not address the implicit information exchange through the status of covert information channels [34]. Such channels usually have rather limited bandwidth for information exchange, and we have not seen any Internet worm that utilizes system timer/clock, CPU utilization, disk space availability, or other covert channels to affect other processes. Therefore, we do not address them in this paper.

We point out that runtime color diffusion is the key difference between process coloring and the log-based tools that are not provenance preserving [24], [25], [31]. Color diffusion propagates the worm break-in provenance information (that is, the color) along the OS information flows so that the *transitive* influence of the worm break-in is captured and recorded in log entries. The three key capabilities of process coloring—color-based worm warning, break-in point identification, and log partitioning—are enabled by provenance preservation. On the other hand, with no provenance information in the log entries, the other log-based tools rely on an external detection point to trigger a worm investigation. Moreover, to identify the break-in point, a back-tracking session needs to be performed using the entire log file as input.

An example: the Slapper worm. Fig. 2 illustrates process color diffusion during the break-in of the Slapper worm [39], which exploits a vulnerable Apache service as its break-in point. In Fig. 2, an oval represents a running process, a rectangle represents a file, and a diamond represents a network socket. Inside the oval are the PID and name of the process. Initially, all Apache *httpd*

1. As shown in [12], to determine whether the information *really* influences the process—*without* the source code of the latter—is equivalent to solving the Halting Problem [44]. To be conservative, we consider that once a process reads from a tainted source, it will be tainted.

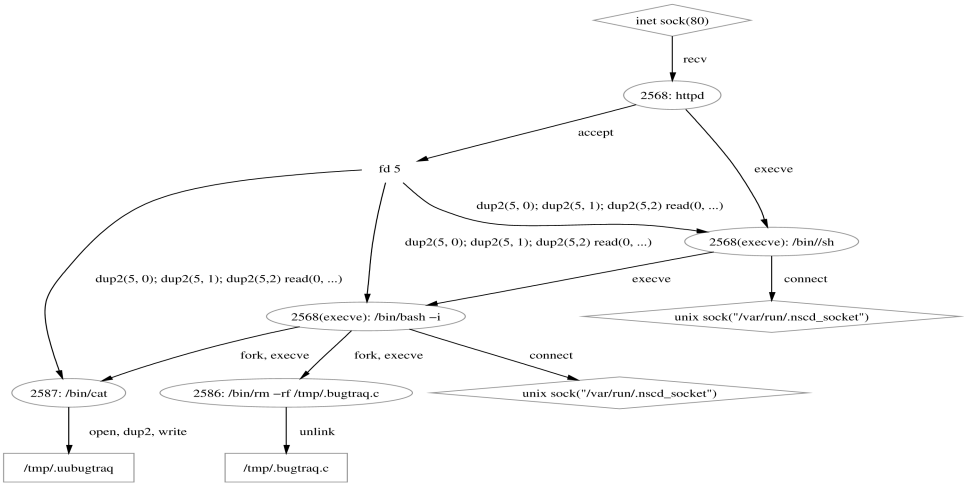


Fig. 2. A process color diffusion example illustrating the break-in of the Slapper worm.

processes are colored "RED." Right after the successful exploitation, the exploited *httpd* process (PID: 2568, color: RED) executes (by *sys_execve* syscall) the program "/bin//sh" (2568, RED), which then executes (by *sys_execve*) 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 color 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 4.1.3, this file will be used to generate (by *sys_read*) the worm process to infect other vulnerable hosts. Via *indirect color diffusion*, the worm process will also be colored "RED."

Theoretical background. Process color diffusion is an instantiation of the generic *label propagation model* [11]. In this model, a system is comprised of active principals and passive objects. Audit information, defined as labels, is propagated according to the information exchange between principals—either directly or indirectly via passive objects—in the system. The key idea is that if one principal causes the information flow [17] of another principal, then the former's labels should be propagated to the latter. We instantiate the label propagation model in the context of process color diffusion along OS-level information flows, starting with the following definitions:

- C: the set of colors initially assigned to service processes as provenance information.
- P: the set of processes (principals) in the host.
- P_g: the subset of processes that are *initially* colored, each of which is a potential worm break-in point.
- O: the set of system objects in the host.
- $init\_color() : C \rightarrow 2^{P_g}$: the initial coloring function assigning a color to a subset of processes $\subset P_g$.[2]

We also define the initial system state S_0 as the state right after initial coloring, where $P_g \neq \emptyset, \forall c, c' \in C : init\_color(c) \cap init\_color(c') = \emptyset$, and $\forall p \in P_g : color(p) \subset C$ ($color()$ is the color set of a principal or an object, as shown in Table 1). The following two properties, which have been proved under the general model [11], also hold in the context of process coloring:

Property 1. *If information is exchanged between principal $p \in init\_color(c)$ and principal p', then c will be in the color set of p' after the information exchange.*

Property 2. *If a color c is found in the color set of principal $p' \notin P_g$, then information was potentially exchanged between p' and principal $p \in init\_color(c)$.*

2.3 Log Collection and Monitoring

Log collection and coloring. Process coloring employs syscall interception to generate log entries and tag them with process colors. As demonstrated in [4], [5], [24], [25], [31], [35], and [41], syscall interception is effective in revealing and understanding intrusion steps and actions. Unfortunately, the commonly used syscall hooking technique (for example, in [4], [27], and [31]) is vulnerable to a *rehooking* attack, where an intruder easily subverts the log collection function [19]. Instead, our process coloring prototype is based on the VMI technique [23], where the interception of syscalls occurs *not* in the syscall dispatcher but *on the virtualization path* of a virtual machine (VM). As such, the interceptor is an integral part of the underlying VM implementation. With log generation, coloring, and storage all taking place outside of the VM, process coloring achieves stronger tamper resistance than existing techniques.

Each log entry will record all the "context" information of a syscall (for example, the current process, syscall number, parameters, return value, return address, and time stamp), which is tagged with the color(s) of the current process. We note that the log format can be easily extended to include richer auditing information such as "who did it" (UID) and "where did it come from/go to" (IP/port).

Real-time log monitoring and warning generation. Process coloring provides a unique opportunity to *externally* monitor the VM without interfering with the VM's normal operations. More specifically, by monitoring the log entries generated at runtime, it is possible to detect anomalies inside the VM caused by worm activities. In particular, the

2. It is possible that more than one process belonging to the same server application be initially assigned the same color.

color(s) of a log entry, combined with other information in the log entry, may reveal the *abnormal influence* between processes that is not supposed to happen under normal circumstances. Such a color-based anomaly will raise a worm warning in real time, which triggers a timely log-based investigation. The following are two examples of a color-based anomaly:

- *Color mixing* is the situation where a previously unicolored process starts to exhibit more than one color. Based on the rationale of color diffusion, color mixing indicates that the process has been influenced by another process with a *different provenance.* Considering the initial assignment of colors to mutually unrelated service processes, such cross-service influence is likely an anomaly and warrants a warning for administrator attention.
- *Unusual color inheritance* is the situation where a process inherits the color of an unlikely parent process. Without color information, this child process (for example, a shell or a utility process like *gcc*, *nice*, and *find*) may look perfectly "normal." However, its color reveals the suspicious *context* under which it is created and therefore raises a warning.

Specific instances of the above color-based warnings will be presented in Section 4.1. They are generated by a real-time *log monitor* running outside of the VM. In addition to the "color-mixing" and "unusual-color-inheritance" anomalies, the administrator will be able to specify more complex or customized anomaly predicates that combine the color information with information in other fields of the log entries.

3 Process Coloring Prototype Implementation

In this section, we present key aspects of the process coloring implementation. Our prototype leverages 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. To support process coloring, a number of key data structures (for example, $task\_struct$ and $ext2\_inode\_info$) 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) $task\_struct$ in the Linux kernel. To facilitate the setting and retrieval of the *color* field, two additional syscalls (*sys_setcolor* and *sys_getcolor*) are implemented. There exists a possibility that these two new syscalls might be abused to undermine process coloring. If their interfaces are exposed, it would be easy for worm authors to add code to corrupt the color assignment. Although a strong authentication scheme may be used to restrict the usage of these two syscalls, it may not be 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 calls only after a service process starts but before it accepts service requests.

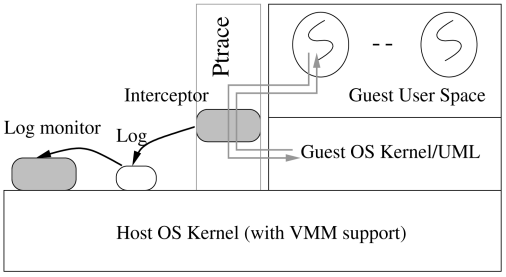


Fig. 3. Tamper-resistant log collection by positioning the interceptor on the syscall virtualization path.

3.2 Color Diffusion

Direct diffusion. If a new process is created by the *fork/vfork/clone* syscall, it will inherit the color of its parent process. When a process is being manipulated via the *ptrace* syscall, the diffusion of color will depend on the syscall 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 syscalls (*sys_waitpid* and *sys_wait4*) that will lead to color diffusion from the child process to the parent process.

Indirect diffusion. Indirect 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 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 the current inode data structure, that is, $ext2\_inode\_info$, we find that the field i_file_acl is intended to record the corresponding access control flags (ACLs) but is *not* used in the *ext2* file system. In our current prototype, this field is leveraged to save the color value (represented as a bitmap) of the corresponding file or directory. For nonpersistent resources (for example, IPC and network sockets), our current prototype only supports sockets, shared memory, and pipes.

3.3 Log Collection and Monitoring

The log collection and coloring mechanism is based on the underlying VM implementation, that is, UML, as shown in Fig. 3. 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 syscalls made by any process in the VM and redirect them to the guest OS kernel. The interceptor for syscall log collection and coloring is located on the syscall virtualization path. Therefore, it is tamper

TABLE 2
Statistics of Log Files Generated by Process Coloring in Three Worm Experiments

	Lion Worm	*Slapper Worm*	*SARS Worm*
Exploited Service (break-in point) (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)
Log collection time period	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-related entries	48.7%	65.9%	12.1%

resistant against malicious processes running *inside* the VM. Moreover, once the interceptor has collected a certain amount of log data (for example, 16 Kbytes), the log data will be pushed down to the host domain. The log file is accessed by a real-time log monitor running in the host domain. The log monitor accepts color-based anomaly predicates specified by the administrator and generates worm warnings at runtime.

4 Experimental Evaluation

4.1 Experiments with Real-World Worms

We evaluate the effectiveness of process coloring using a number of real-world Internet worms including Adore [1], Ramen[3] [2], Lion [3], Slapper [39], SARS [8], and their variants. Each worm experiment is conducted in a virtual distributed worm playground called *vGround* [29], which is a realistic, confined, and scaled-down network environment. A vGround consists of network entities and end hosts both realized as VMs. The end-host VMs are enhanced with process coloring. In our worm experiments, the vGrounds involve server VMs running real-world services, as well as client VMs running as service requestors. Meanwhile, vGround strictly confines worm traffic and damages for experiment safety.

4.1.1 Efficiency of Worm Investigations

We present in detail the experiments with Lion, Slapper, and SARS worms to demonstrate the three new capabilities of process coloring (Section 1). The color-based worm warning capability will be described in the next three sections. In this section, we focus on the efficiency of worm investigations enabled by process coloring. Table 2 shows the key statistics of the respective log files created in the worm experiments. Each log file contains log entries collected during a 24-hour period, including both worm-related entries and normal service access entries.

During each experiment, we are able to name the worm's break-in point (second row in Table 2) readily by the color of the log entry involved in the corresponding worm warning. On the other hand, the non-provenance-preserving tools [24], [25], [31] will have to perform a traceback using the entire log file as input. To reconstruct the contamination actions of the worm, only the log partition that bears the color of the break-in point needs to be processed, whereas the entire log file is needed by the non-provenance-preserving tools. More specifically, the log partition contains 48.7 percent (Lion worm), 65.9 percent (Slapper worm), and 12.1 percent (SARS worm) of the log entries in the respective log file (last row in Table 2). We point out that because log entries are naturally partitioned by their colors, increasing requests to other unexploited services in the experiments will further lower the percentage of worm-related log entries.

4.1.2 Lion Worm Experiment

Experiment setup. Fig. 4 shows a process coloring view of an *uninfected* server VM that hosts a number of services: a BIND (bind-8.2.2_P5-9) service, an NFS/RPC service (*portmap* and *rpc.statd*), a printer service (*lpd*), and a Web server (*ghttpd*). Each service is assigned its own color. In particular, the BIND (DNS) service vulnerable to the Lion worm is assigned the color "RED." From another VM in the vGround, we launch the Lion worm to infect the server VM.[4] Before the worm infection, we, as the administrator, set two worm warning predicates in the external real-time log monitor. The first predicate states that an "unusual-color-inheritance" warning will be raised if a log entry is generated by a *shell* process that inherits the color of the BIND service ("RED"). The second predicate states that a "color-mixing" warning will be raised if a log entry is generated by one of the service processes that bears more than its original color.

Color-based warning and break-in point identification. After the Lion worm attack begins, the real-time log monitor raises two warnings. The first warning ("unusual color inheritance") is triggered by a log entry generated by a process *sh*:

```
RED: 31168 ["sh"]: 11_execve("/bin/rm -rf
dev/.lib")
```

This is not a normal shell process as it bears the color ("RED") of BIND. In normal operations, the BIND service is not supposed to create or influence a shell process.

The second warning ("color mixing") is triggered by a log entry generated by the *ghttpd* Web server when making

3. The 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 break-in.

4. This "seed" worm is instrumented to target the vulnerable server VM. However, the worm copy injected to the server VM is of the original version.

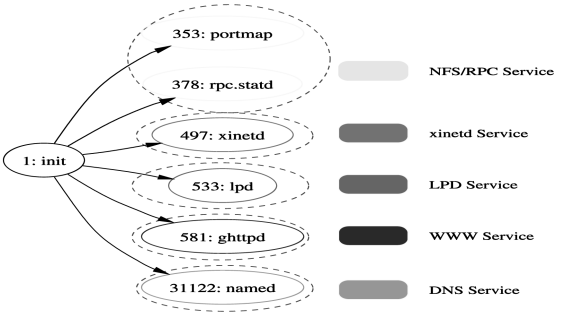


Fig. 4. A process coloring view of a vulnerable server VM *before* Lion infection.

a *read* call (the entry is not shown due to its extreme length). It shows that the *ghttpd* process, originally assigned "NAVY," suddenly bears both "NAVY" and "RED" colors. This is suspicious because the Web server is not supposed to be influenced by the BIND service. Detailed explanation for this color mixing will be described in the reconstruction of Lion contaminations.

Both warnings are real-time detection points that lead to the investigation of the Lion worm break-in and contaminations. Before a detailed log analysis, the "RED" color readily indicates that the break-in point is the BIND service—an improvement in investigation efficiency over the non-provenance-preserving tools.

Color-based log partitioning. With the log file partitioned by colors, only the "RED" log entries are used as input for the reconstruction of Lion worm contaminations. The result is shown in Fig. 5. We note that the procedure of log-based contamination reconstruction itself is not performed by process coloring. Instead, we can simply leverage the causality-based back-tracking [31] and forward-tracking [33] algorithms—with a reduced input size.

We describe the result in Fig. 5 to demonstrate the *sufficiency* of color-based log partitioning. In other words, we show that by using only the "RED" log entries, we can still derive a complete account of Lion contamination actions, and we confirm this by comparing our result with a detailed Lion worm report [3]. In Fig. 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 Fig. 5) to the server VM. The worm overwrites all HTML files named *index.html* in the system with its own HTML file for Web defacement (circle 3). Interestingly, we observe from the log that the worm attempts to execute the file replacement *twice*—a detail *not* reported in [3]. The first file replacement attempt is by 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). Recall the color-mixing warning at runtime—it is caused by the *index.html* file replacement. As soon as the *ghttpd* process ("NAVY") reads the replaced file ("RED"), the color mixing occurs.

The worm then tries to initiate the next round of infection (circle 4). In the thickly 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.[5] Since there is only one server VM running the vulnerable BIND service in the vGround, the worm cannot find another host to infect, and the file *bindname.log* storing the IP addresses of possible victims remains empty.

4.1.3 Slapper Worm Experiment

Experiment setup. The Slapper worm experiment is conducted in a different vGround. We initially assign various colors to service processes in an uninfected server VM. Especially, the vulnerable Apache service (apache-1.3.19-5 with openssl-0.9.6b-8 package) is assigned "RED." Through direct diffusion, the colors of all spawned httpd worker processes are also "RED." A process coloring view of the server VM *before* the Slapper infection is shown in Fig. 6. The normal Web requests are generated by multiple client VMs requesting a 2,890-byte *index.html* file.

Color-based warning and break-in point identification. After the Slapper worm attack begins, the real-time log monitor raises two "unusual-color-inheritance" warnings. The first warning is an abnormal *sh* process bearing the color ("RED") of the Apache service:

```
RED: 2563 ["sh"]: 11_execve("/bin/bash -i")
```

The second warning is caused by a "RED" *gcc* process:

```
RED: 2586 ["gcc"]:
11_execve("/usr/lib/gcc-lib/i386-redhat-
   linux/2.96/cpp0    -lang-c
-D__GNUC__=2   -D__GNUC_MINOR__=96
-D__GNUC_PATCHLEVEL__=0  -D__ELF__   -Dunix
-Dlinux  -D__ELF__   -D__unix__
-D__linux__   -D__unix   -D__linux
-Asystem(posix)   -Acpu(i386)
-Amachine(i386)   -Di386  -D__i386
-D__i386__   -D__tune_i386__
/tmp/.bugtraq.c   /tmp/cc0f78Vl.i")
```

Under normal circumstances, these processes are not likely to be spawned by a Web server. In addition to the above two warnings, we also notice from the real-time log monitor that there is a surge of "RED" log entries ($>$ 10,000 in 1 minute) generated *without* a corresponding Web request rate increase. This is also suspicious as a normal Web access generates only 15 log entries that represent the known sequence of Apache server actions.

All the above real-time anomalies warrant further investigation of the "RED" log entries. Before a detailed log analysis, we are able to determine that the break-in point of the attack is the Apache Web server.

Color-based log partitioning. The "RED" log entries constitute 65.9 percent of the entire log file. The relatively high percentage of "RED" entries is a result of the large distinct footprint of the Slapper worm in the victim. During the transmission of a Slapper worm copy, a *uuencoded* source file is sent from the infecter to the victim. More specifically, the sender issues a *sendch* call for *every byte* of the uuencoded file. Correspondingly, the victim calls *sys_read* for every byte received (totaling 94,320 calls).

5. A forensic analysis of the VM reveals that these two files contain the exploitation code for the BIND vulnerability.

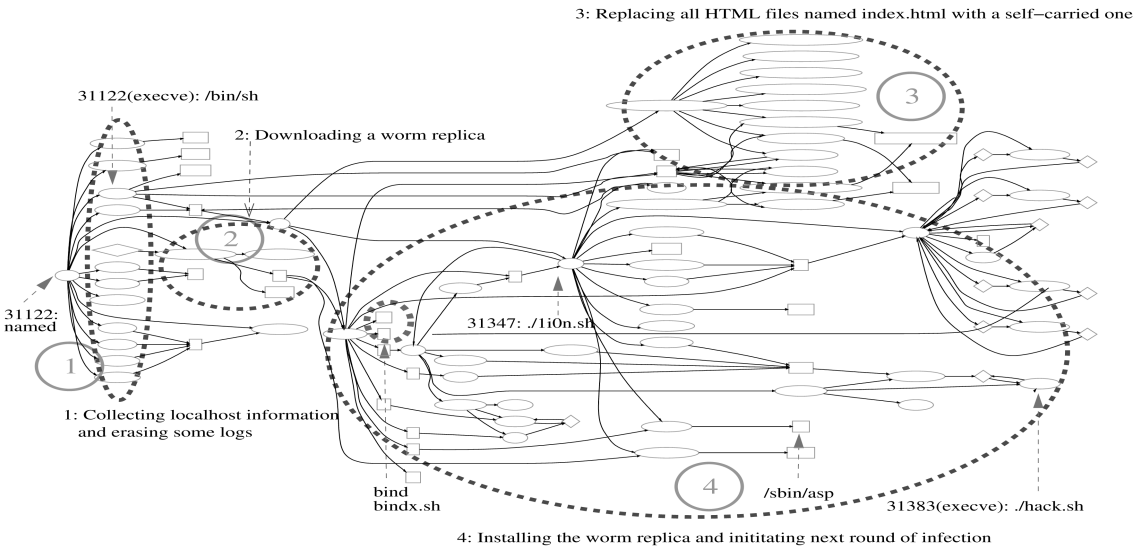


Fig. 5. Lion worm contaminations reconstructed from "RED" log entries.

Moreover, each encoded byte is then written (using the *cat* command) to a local file named */tmp/.uubugtraq*, leading to another 94,320 *sys_write* syscalls. The result of the Slapper worm contamination reconstruction is shown in Fig. 7.

To show the sufficiency of "RED" log entries, we compare our result with a detailed Slapper worm analysis [39] and confirm that Fig. 7 reveals all contaminations by the Slapper worm. The worm first exploits an httpd worker process (PID:2568) to gain system access. A uuencoded version of the worm source code is then downloaded (circle 1 in Fig. 7) and *uudecoded* (circle 2) to reconstruct the original code, which is compiled (circle 3) to generate the worm binary code. The binary code is executed (circle 4) in an attempt to infect other hosts.

Interestingly, Fig. 7, constructed by the causality-based algorithm [31], [33], does not reveal a *preamble* of the Slapper worm attack. This preamble can be uncovered by searching the "RED" log entries as follows: First, the IP address of the infecter VM can be derived from the log entry that records the first *accept* syscall in Fig. 7. This IP address is then searched against the "RED" log entries not involved in Fig. 7. There are 23 log entries found containing the same IP address, which record 23 *accept* calls made right *before* the actual Slapper exploitation takes place. These calls correspond to 23 TCP connections initiated from the (same) infecter VM: the first 21 connections have *no* payload, the 22nd connection is an invalid HTTP request, which turns out to be a request to obtain the Apache server version, and the 23rd connection leads to a short interaction, as shown in the log excerpt in Fig. 8. From [39], we know that the 21 no-payload connections are for checking the reachability of the Apache server and for depleting the Apache server pool to make sure that the two subsequent exploitations have the same heap layout. The preexploitation aims at deriving the overwritable heap address in the vulnerable Apache server. This heap address is then reused in the actual exploitation.

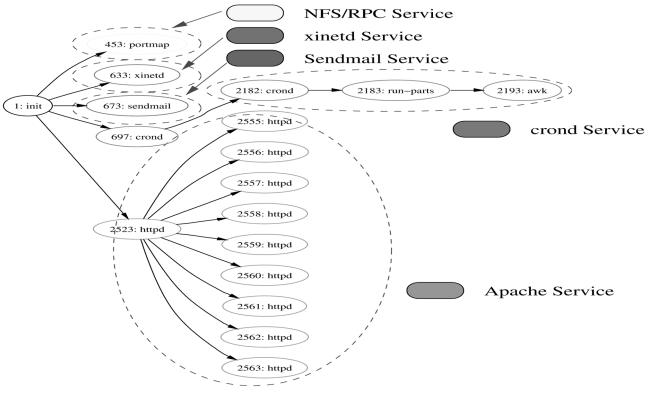


Fig. 6. A process coloring view of a vulnerable server VM *before* Slapper infection.

4.1.4 SARS Worm Experiment

Experiment setup. The SARS worm is a multiplatform worm that 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 VMs, our experiment is conducted in a Linux-based vGround. The vulnerable Samba service (samba-2.2.5-10) is assigned "RED."

Color-based warning and break-in point identification. Similar to the Lion worm experiment, the real-time log monitor issues two warnings: One is an "unusual-color-inheritance" warning involving an abnormal *sh* process bearing the color of the Samba server ("RED"), whereas the other is a "color-mixing" warning where the originally "YELLOW" *sendmail* service suddenly acquires the color of a Samba service ("RED"). These two warnings warrant a detailed log analysis, before which we can readily infer that the break-in point is the Samba service.

Color-based log partitioning. The "RED" log entries account for 12.1 percent of the entire log file. Fig. 9 shows the Redhat-8.0-based server VM *after* the SARS infection. The figure indicates that the exploitation code involves some redundancy as two "RED" */bin//sh* processes are executed: One quits immediately after its creation, whereas

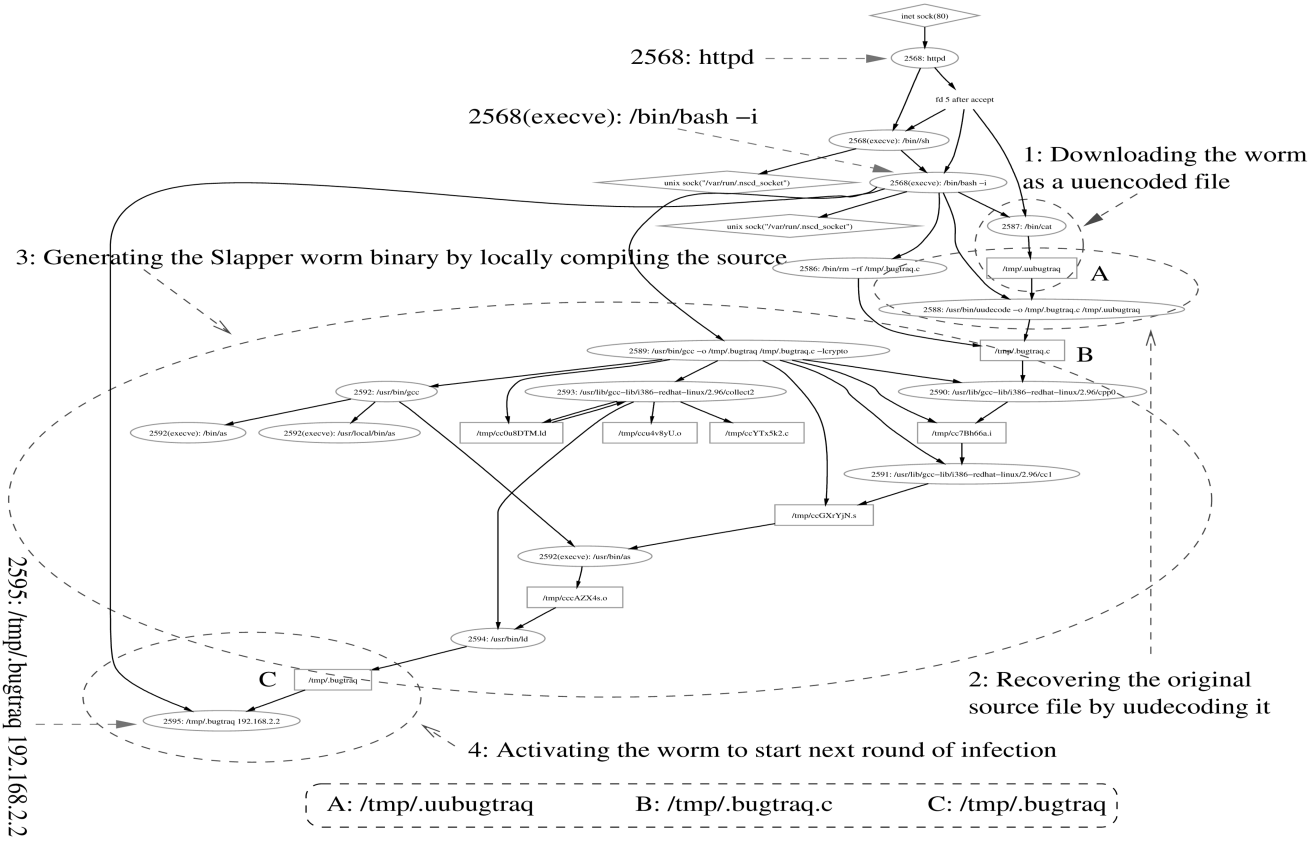


Fig. 7. Slapper worm contaminations reconstructed from "RED" log entries.

the other one retrieves the worm copy and starts process *start.sh* (PID: 6285, shown in Fig. 9), which executes the worm binary in the victim. From the "RED" entries, the full account of SARS worm contaminations can be reconstructed. We observe that the SARS worm contains a primitive user-level *rootkit* with the purpose of hiding the existence of worm-related files, directories, active processes, and network connections. The SARS worm also inserts a number of backdoors such as a Web server and an ICMP-based backdoor, which allow an attacker to access the infected host later. Systemwide information such as the host's IP address and the information in configuration files */etc/hosts* and */etc/passwd* is collected by the worm and sent to a hard-coded e-mail account as an e-mail attachment. Note that this is the reason for the earlier "color-mixing" warning when the "YELLOW" *sendmail* is tainted "RED"—the color of Samba. The equipment of advanced payloads, such as the rootkit in the SARS worm, indicates the recent trend of increasingly stealthy worms in the making.

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

Fig. 8. Log excerpt showing the preexploitation of the Slapper worm attempting to get the overwritable heap address in the vulnerable Apache server. BROWN log entries are not related.

4.2 Runtime Overhead of Process Coloring

To measure the system overhead introduced by process coloring, we perform a number of benchmarking experiments using McVoy and Staelin's LMbench [36], 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 2.6-GHz Intel Xeon processor and 2 Gbytes of 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 capabilities (COLORING). The results are shown in Table 3.

Table 3a shows the process operation overhead. Table 3b shows the context switching time under varying numbers of processes and working set sizes. The file system and virtual memory latency results are shown in Table 3c. The results indicate that UML suffers a significant performance penalty caused by its user-level implementation. However, process coloring only incurs a small *extra* performance degradation

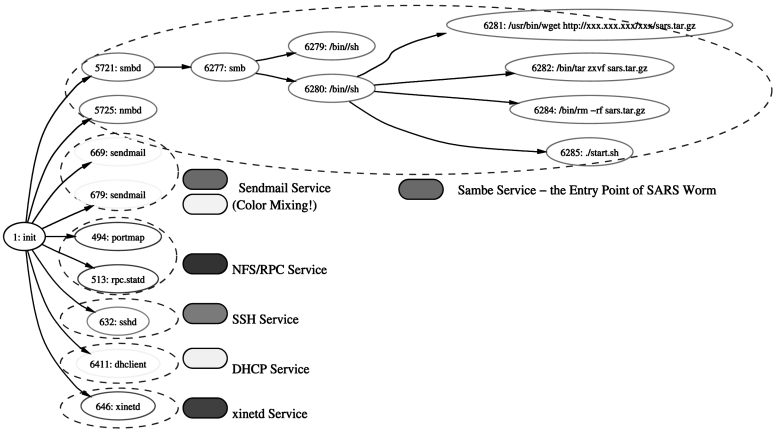


Fig. 9. A process coloring view of a Redhat 8.0 server running multiple services *after* it is infected by the SARS worm. The anomalous color mixing in the *sendmail* process triggers the SARS worm investigation.

on top of that from the original UML. The reason lies in the interceptor placement. By positioning the interceptor *within* the syscall virtualization path, our prototype is able to avoid an additional context switch per syscall, which is needed in other syscall interception schemes [35]. In addition, the log data push-down is not performed upon every invocation of a syscall. Instead, an internal cache (16 Kbytes) 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) will be significantly reduced with more efficient VM platforms (for example, Xen [20] via paravirtualization) and the upcoming architecture support for VMs (for example, Intel's Vanderpool technology [22]).

5 Discussion

In this section, we examine possible evasion strategies, as well as a limitation of process coloring.

Low-level evasion. The integrity of the colors of active processes and intermediate resources is critical to the trustworthiness of worm investigation results. Since the current prototype maintains the color information within the kernel of the system being monitored, it is possible that this information may be manipulated through certain low-level attacks. For example, if the process color is recorded in the *task_struct* PCB structure, a method called direct kernel object manipulation (DKOM) [13] can be used to modify the color value (for example, by writing to the special device file */dev/kmem*). Fortunately, solutions such as CoPilot [40], Livewire [23], and Pioneer [42] have been proposed to address the issue of OS kernel integrity. Another possible countermeasure is to create a *shadow* structure, which is maintained by the VMM and is thus inaccessible from inside the VM. However, compared with our current prototype, the shadowing solution poses a significantly greater challenge in deriving VM operation semantics from low-level information collected via VMI, affecting the accuracy and completeness of worm investigation results.

TABLE 3
LMBench Results Showing Low *Additional* Process Coloring Overhead

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
COLORING	11.0	147	29.0	4910	8221

(a)

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
COLORING	10.9	11.5	10.7	19.1	47.2

(b)

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
COLORING	228.6	90.2	792.0	15.1	21.9

(c)

(a) Process-related times in μs. (b) Context switching times in μs. (c) File and VM system latencies in μs.

Evasion by diffusion cutting. It is possible that a worm uses a hidden channel to escape color diffusion. For example, a worm could use the initial part of an attack to crack a weak password, which is later used in a *separate* session to gain system access and complete the rest of the 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 during the second break-in will 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.

Evasion by color saturation. If a worm is aware of the coloring scheme, it may attempt to acquire more colors or introduce many "noise" log entries from unrelated services right after its break-in. As a result, the associated colors cannot uniquely identify the break-in point. However, to the worm's *disadvantage*, the color saturation attack will lead to the *color-mixing* anomaly, which gives away the worm immediately. The 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 of a process (using a bitmap), 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 their own or others' actions. With this distinction, the inherited colors can be used to partition the log file, whereas the diffused colors can be used to detect a color-mixing point for further examination of other log partitions possibly affected.

We point out that, like other anomaly-based detection techniques, the color-based warning capability could lead to false positives. However, our experiments have so far produced few false positives in consolidated server environments. This can be explained by the nature of color-based anomalies, which indicates the abnormal influence or dependency between processes that are supposed to be independent of each other. Considering the processes that originate from various independent server applications, any unexpected influence or dependency between them is deemed anomalous and will be captured by a color-based warning. We do acknowledge that in other operating environments not addressed in this paper, the color-based warning capability could result in a higher false-positive rate. For example, a client machine, different from a server host, may run client-side applications with a legal influence or dependency between each other (for example, via shared files or common helper processes). In this environment, process coloring may generate a false alarm when a legal interapplication influence is taking place, and eliminating such false positives is our ongoing work. Our expectation is that such false positives can be reduced by enhancing the color-based anomaly predicates to specify—and thus to exclude—a legal interapplication influence. The enhanced predicates will leverage the rich semantic contextual information carried by the log entries. Another effective approach to reducing false positives is to "insulate" the few shared files and helper processes so that the interapplication influence caused by these common objects/subjects will not raise color anomaly alarms.

6 Related Work

Process coloring can be integrated into existing log-based intrusion investigation tools [24], [25], [31], [33], making them provenance preserving. Most notably, both BackTracker [31] and Taser [25] are able to reconstruct the sequence of steps that have occurred during an intrusion—from an external detection point (for example, a corrupted file) back to the break-in point. The forward-tracking extension [33] of BackTracker further identifies all possible damages caused by the intrusion after the back-tracking session. Both back-tracking and forward-tracking require the entire log as input. With process coloring, the break-in point can be determined or narrowed down by the color(s) of the detection point, whereas the volume of the inspected log can be reduced by color-based log partitioning. Moreover, process coloring brings the new capability of real-time worm warning by detecting color-based anomalies, turning the log file into an active warning generator.

Information flow models [10], [16], [17] have been increasingly applied as the underpinnings of taint-based security techniques developed at different levels, including the instruction [14], [38], language [28], [37], [47], and OS (this work, [25], and [31]) levels. These techniques complement each other and are applicable to different scenarios. TaintCheck [38] works at the instruction level to detect overwrite attacks and generate exploit signatures. TaintBochs [14] focuses on the lifetime tracking of sensitive data (for example, passwords) stored in the memory. Both of them monitor information flows at the granularity of machine instruction and therefore are not able to provide semantic information about systemwide worm contaminations (for example: What are the commands executed by the worm? Which files are affected by the worm? Are there backdoors or rootkits in the compromised system?). The language-level techniques [28], [37], [47] track information flows at the granularity of object, variable, or memory location *inside* a program. As a result, they are able to leverage fine-grain application semantic information to detect attacks against the program (which usually requires its source code). However, they cannot capture the influence *between* OS-level processes, which is the main focus of process coloring. Finally, process coloring does not require the program source code, making it suitable for the first line of worm defense.

Process coloring can also be applied to file and transaction repair/recovery systems. The repairable file service [48] aims at identifying 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 causes a human 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, a similar technique [9] exists in the database area, which is capable of recording contaminations at the transaction level and rolling

back the damages if a 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 suspicious log colors and achieving more tamper-resistant log collection.

Recent advances in VM technologies have created tremendous opportunities for intrusion monitoring and replay [5], [21], [23], [30], system problem diagnosis [32], [45], [46], attack recovery and avoidance [21], [43], and data lifetime tracking [14], [15]. For example, ReVirt [21] is able to replay a system's execution at the instruction level. Time-traveling VMs such as [32], [45], and [46] provide highly effective means to reexamine and troubleshoot the system execution and configuration. Process coloring complements these efforts by leveraging VM technologies for worm warning, break-in point identification, and log partitioning. In addition, process coloring can be integrated into other VM-based networked server systems to add provenance awareness to these systems.

7 Conclusion

We have presented the design, implementation, and evaluation of process coloring, a provenance-preserving approach to worm warning and investigation. By associating a unique color to each remotely accessible service and diffusing the color based on actions performed by processes in a networked server host, process coloring preserves the worm break-in provenance information and propagates it along OS information flows. Process coloring achieves three key capabilities: 1) color-based worm warning, 2) color-based determination of worm break-in points, and 3) color-based log partitioning to reduce the input size of worm contamination reconstruction. The VMI-based implementation enables external log collection, storage, and monitoring. Our experiments with a number of real-world Internet worms demonstrate the efficiency and effectiveness of process coloring.

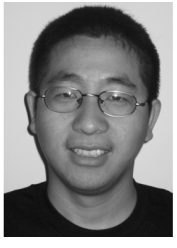
Acknowledgments

The authors would like to thank the anonymous *IEEE Transactions on Parallel and Distributed Systems (TPDS)* reviewers whose comments have helped to improve the presentation of this paper. The anonymous reviewers of a preliminary conference version of this paper [49] are also acknowledged. This work was supported in part by a gift from Microsoft Research and by the US National Science Foundation (NSF) under Grants OCI-0438246, OCI-0504261, and CNS-0546173. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] *Linux Adore Worms,* http://securityresponse.symantec.com/avcenter/venc/data/linux.adore.worm.htm, 2007.
[2] *Linux Ramen Worm,* http://service1.symantec.com/sarc/sarc.nsf/html/pf/linux.ramen.worm.html, 2007.
[3] *SANS Institute: Lion Worm,* http://www.sans.com/y2k/lion.htm, 2007.
[4] *Sebek,* http://www.honeynet.org/tools/sebek/, 2007.
[5] *The Honeynet Project,* http://www.honeynet.org, 2007.
[6] *The Strange Decline of Computer Worms,* http://www.theregister.co.uk/2005/03/17/f-secure_websec/print.html, 2007.
[7] *Virus Writers Get Stealthy,* http://news.zdnet.co.uk/internet/security/0,39020375,39191840,00.htm, 2007.
[8] *SARS Worms,* http://www.xfocus.net/tools/200306/413.html, June 2003.
[9] P. Ammann, S. Jajodia, and P. Liu, "Recovery from Malicious Transactions," *IEEE Trans. Knowledge and Data Eng.,* vol. 14, no. 5, pp. 1167-1185, Sept. 2002.
[10] D. Bell and L. LaPadula, "MITRE Technical Report 2547 (Secure Computer System): Volume II," *J. Computer Security,* vol. 4, nos. 2/3, pp. 239-263, 1996.
[11] F. Buchholz, "Pervasive Binding of Labels to System Processes," PhD dissertation, Purdue Univ., also as CERIAS Technical Report 2005-54, 2005.
[12] F. Buchholz and E.H. Spafford, "On the Role of File System Metadata in Digital Forensics," *J. Digital Investigation,* Dec. 2004.
[13] J. Butler, *Direct Kernel Object Manipulation (DKOM),* http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf, 2004.
[14] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation," *Proc. 13th Usenix Security Symp.,* Aug. 2004.
[15] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation," *Proc. 14th Usenix Security Symp.,* Aug. 2005.
[16] D.R. Clark and D.R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," *Proc. IEEE Symp. Security and Privacy (S&P '87),* pp. 184-194, 1987.
[17] D.E. Denning, "A Lattice Model of Secure Information Flow," *Comm. ACM,* vol. 19, pp. 236-243, May 1976.
[18] J. Dike, *User Mode Linux,* http://user-mode-linux.sourceforge.net, 2007.
[19] M. Dornseif, T. Holz, and C. Klein, "NoSEBrEaK—Attacking Honeynets," *Proc. Fifth Ann. IEEE Information Assurance Workshop,* June 2004.
[20] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the Art of Virtualization," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP '03),* Oct. 2003.
[21] G.W. Dunlap, S.T. King, S. Cinar, M.A. Basrai, and P.M. Chen, "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay," *Proc. Fifth Symp. Operating Systems Design and Implementation (OSDI '02),* Dec. 2002.
[22] R. Uhlig et al., "Intel Virtualization Technology," *Computer,* special issue on virtualization technology, May 2005.
[23] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," *Proc. Network and Distributed System Security Symp. (NDSS '03),* Feb. 2003.
[24] A. Goel, W.-C. Feng, D. Maier, W.-C. Feng, and J. Walpole, "Forensix: A Robust, High-Performance Reconstruction System," *Proc. Second Int'l Workshop Security in Distributed Computing Systems (SDCS '05),* June 2005.
[25] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The Taser Intrusion Recovery System," *Proc. 20th ACM Symp. Operating Systems Principles (SOSP '05),* Oct. 2005.
[26] J.A. Goguen and J. Meseguer, "Security Policies and Security Models," *Proc. IEEE Symp. Security and Privacy (S&P '82),* pp. 11-20, 1982.
[27] J. Grizzard, J. Levine, and H. Owen, "Re-Establishing Trust in Compromised Systems: Recovering from Rootkits that Trojan the System Call Table," *Proc. Ninth European Symp. Research in Computer Security (ESORICS '04),* Sept. 2004.
[28] V. Halder, D. Chandra, and M. Franz, "Practical, Dynamic Information Flow for Virtual Machines," *Proc. Second Int'l Workshop Programming Language Interference and Dependence (PLID '05),* 2005.
[29] X. Jiang, D. Xu, H.J. Wang, and E.H. Spafford, "Virtual Playgrounds for Worm Behavior Investigation," *Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID '05),* Sept. 2005.
[30] X. Jiang and D. Xu, "Collapsar: A VM-Based Architecture for Network Attack Detention Center," *Proc. 13th Usenix Security Symp.,* Aug. 2004.

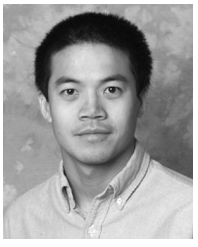
[31] S.T. King and P.M. Chen, "Backtracking Intrusions," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP '03),* Oct. 2003.
[32] S.T. King, G.W. Dunlap, and P.M. Chen, "Debugging Operating Systems with Time-Traveling Virtual Machines," *Proc. Usenix Ann. Technical Conf.,* Apr. 2005.
[33] S.T. King, Z.M. Mao, D.G. Lucchetti, and P.M. Chen, "Enriching Intrusion Alerts through Multi-Host Causality," *Proc. Network and Distributed System Security Symp. (NDSS '05),* Feb. 2005.
[34] B. Lampson, "Protection," *Proc. Fifth Princeton Conf. Information Sciences and Systems,* pp. 437-443, 1971.
[35] Z. Liang, V.N. Venkatakrishnan, and R. Sekar, "Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs," *Proc. 19th Ann. Computer Security Applications Conf. (ACSAC '03),* Dec. 2003.
[36] L. McVoy and C. Staelin, "LMbench: Portable Tools for Performance Analysis," *Proc. Usenix Ann. Technical Conf.,* 1996.
[37] A.C. Myers, "JFlow: Practical Mostly-Static Information Flow Control," *Proc. 26th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '99),* 1999.
[38] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Proc. Network and Distributed System Security Symp. (NDSS '05),* Feb. 2005.
[39] F. Perriot and P. Szor, *An Analysis of the Slapper Worm Exploit,* white paper, Symantec, http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf, 2007.
[40] N.L. Petroni, T. Fraser, J. Molina, and W.A. Arbaugh, "Copilot—A Coprocessor-Based Kernel Runtime Integrity Monitor," *Proc. 13th Usenix Security Symp.,* Aug. 2004.
[41] N. Provos, "Improving Host Security with System Call Policies," *Proc. 12th Usenix Security Symp.,* Aug. 2003.
[42] 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," *Proc. 20th ACM Symp. Operating Systems Principles (SOSP '05),* Oct. 2005.
[43] A. Stavrou, A.D. Keromytis, J. Nieh, V. Misra, and D. Rubenstein, "MOVE: An End-to-End Solution to Network Denial of Service," *Proc. Symp. Network and Distributed System Security (NDSS '05),* Feb. 2005.
[44] A.M. Turing, "On Computable Numbers, with an Application to the Entscheidungs Problem," *Proc. London Math. Soc. Series 2,* vol. 42, pp. 230-265, 1937.
[45] A. Whitaker, R.S. Cox, and S.D. Gribble, "Configuration Debugging as Search: Finding the Needle in the Haystack," *Proc. Sixth Symp. Operating Systems Design and Implementation (OSDI '04),* Dec. 2004.
[46] A. Whitaker, R.S. Cox, and S.D. Gribble, "Using Time Travel to Diagnose Computer Problems," *Proc. 11th ACM SIGOPS European Workshop,* Sept. 2004.
[47] W. Xu, S. Bhatkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," *Proc. 15th Usenix Security Symp.,* 2006.
[48] N. Zhu and T. Chiueh, "Design, Implementation and Evaluation of Repairable File Service," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '03).* , June 2003.
[49] X. Jiang, A. Walters, F. Buchholz, D. Xu, Y.M. Wang, and E.H. Spafford, "Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach," *Proc. 26th IEEE Int'l Conf. Distributed Computing Systems (ICDCS '06),* July 2006.



Xuxian Jiang received the BS degree from Xi'an Jiaotong University in 1998 and the PhD degree in computer science from Purdue University in 2006. Since 2006, he has been a faculty member at George Mason University, where he is currently an assistant professor of computer science. His research interests include operating systems security and malware defense. He is a member of the IEEE.

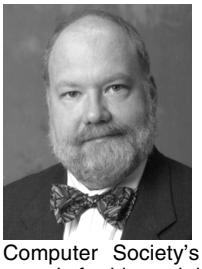
Florian Buchholz received the PhD degree in computer science in August 2005 from Purdue University, where he worked with his adviser, Eugene Spafford. He joined the faculty of the Computer Science Department, James Madison University (JMU), in September 2005 as an assistant professor. He is teaching computer forensics and distributed security classes at JMU as part of the online Master's program with emphasis on information security, as well as various undergraduate classes, including a computer forensics course. His research focuses on digital forensics with emphasis on time synchronization and event and data reconstruction, operating system security, and network traceback. He is a member of the IEEE.

Aaron Walters received the BS degree in computer engineering from the University of Notre Dame in 2001 and the MS degree in computer science from Purdue University in 2006. He is a founding partner of Volatile Systems. Prior to joining Volatile Systems, he was the director of forensics at Komoku, Inc. and the research lead of BAE Systems Advanced Detection Research Group. He was a member of the Center for Education and Research in Information Assurance and Security (CERIAS) and the Dependable and Secure Distributed Systems Laboratory. His research interests include distributed systems, anomaly detection, digital forensics, and multisensor data fusion.



Dongyan Xu received the BS degree from Zhongshan (Sun Yat-Sen) University in 1994 and the PhD degree in computer science from the University of Illinois, Urbana-Champaign, in 2001. Since 2001, he has been a faculty member at Purdue University, where he is currently an associate professor of computer science and electrical and computer engineering (by courtesy). He is also affiliated with the Center for Education and Research in Information Assurance and Security (CERIAS). His research interests include virtualization technologies, distributed systems, and malware defense. He is a recipient of a US National Science Foundation (NSF) CAREER Award.

Yi-Min Wang received the BS degree from National Taiwan University in 1986 and the PhD degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign, in 1993. He worked at AT&T Bell Laboratories from 1993 to 1997 and joined Microsoft in 1998. He is the director and principal researcher at the Internet Services Research Center (ISRC), Microsoft Research-Redmond, where he leads an R&D group responsible for systems, infrastructure, cyber-intelligence, and search quality. His research interests include security, systems management, dependability, home networking, and distributed systems. He is a senior member of the IEEE and the IEEE Computer Society.



Eugene H. Spafford has been with Purdue University since 1987, where he is a professor of computer science. He is also the executive director of the Center for Education and Research in Information Assurance and Security (CERIAS). His current research is directed toward cyber forensic issues and national security policies. He is a fellow of the IEEE, the ACM, and the AAAS. He is a past recipient of the IEEE Taylor Both Award and the IEEE Computer Society's Technical Achievement Award, among other awards for his work in information security and policy.