Recovering from intrusions in distributed systems with DARE

Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich
MIT CSAIL

ABSTRACT

DARE is a system that recovers system integrity after intrusions that spread between machines in a distributed system. DARE extends the rollback-and-reexecute recovery model of Retro [14] to distributed system recovery by solving the following challenges: tracking dependencies across machines, repairing network connections, minimizing distributed repair, and dealing with long-running daemon processes. This paper describes an early prototype of DARE, presents some preliminary results, and discusses open problems.

1 INTRODUCTION

An adversary that compromises one machine in a distributed system, such as a cluster of servers, can often leverage the trust between machines in that system to compromise other machines as well [12]. Recovering the integrity of these machines after an intrusion is often a manual process for system administrators, and this paper presents a system that helps administrators automate intrusion recovery across machines.

To understand the steps involved in intrusion recovery, consider a recent break-in at SourceForge, a source code repository that hosts over 300K open source projects. On January 26th, 2011, system administrators of SourceForge detected a targeted attack that infected multiple machines in their network [9]. A key goal for the administrators was to determine whether any source code or files hosted by SourceForge were modified by the attack, and to restore their integrity, but to do that they needed to first restore the integrity of their servers.

As a first step, SourceForge immediately locked down possibly affected servers, and started investigating logs of their services to determine the root cause of the attack and the extent of the damage it caused. Two days later, SourceForge decided to reset passwords of two million accounts, due to evidence of attempts to sniff user passwords, even though their administrators were unable to determine whether the attempts were successful [10]. Then, they validated project data such as source commits and file releases, by comparing data of the compromised servers with the latest backup data before the attack, and notified project owners if they detected suspicious changes. Finally, they restored services such as SVN, file hosting, and project webspaces, to previously working copies, by completely reinstalling servers. In all, this recovery process required five days of effort by the entire SourceForge team, during which time several SourceForge services were unavailable.

This example shows that in today's distributed systems, recovering from an attack that propagates across machines is manual, tedious, and time-consuming. Worse yet, there is no guarantee that the administrator found all effects of the attack and recovered from them. The SourceForge attack is not an isolated example of these types of attacks; other such high-profile attacks include Stuxnet [13] and the recent `kernel.org` compromise [7]. Given that today's systems are highly interconnected, recovering from such distributed attacks is an important problem. However, past work [12] does not address intrusion recovery in a distributed system.

In this paper, we discuss the design of an early prototype of DARE,[1] a system for recovering from an attack that propagates across a distributed system, once an administrator has identified the source of the attack. At a high level, DARE adopts Retro's [14] rollback-and-reexecute approach to recovery, but additionally also tracks dependency information across machines. Thus, if an attack propagates to other machines, DARE automatically initiates repair on them and eventually undoes the effects of the attack on all the infected machines in the system.

DARE's distributed repair faces four challenges that are not addressed by Retro. First, DARE needs to determine dependencies between events that occur on differ-

[1] DARE stands for Distributed Automatic REpair.

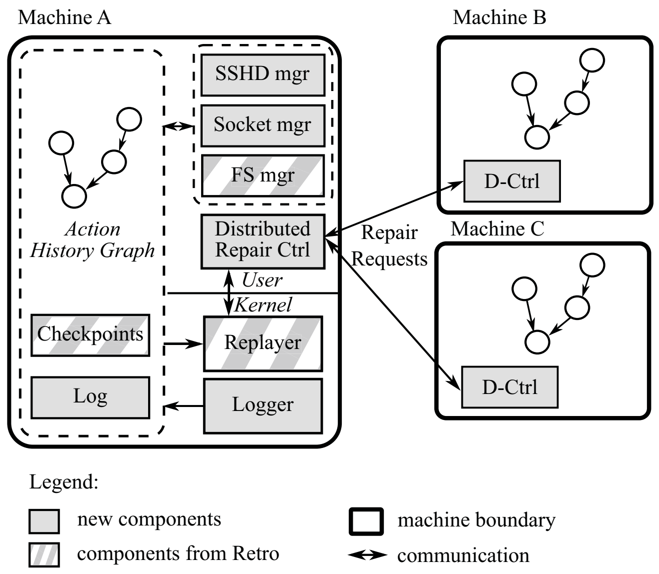

Figure 1: An overview of DARE's design. Components introduced by DARE are shaded. Components borrowed from Retro are striped. During normal execution, the logger interposes on system calls and logs them. The action history graph is generated from this logged information. During repair, the distributed repair controller listens for repair requests from the administrator locally or from remote controllers, and uses its action history graph to perform repair on the local machine and to invoke repair on remote controllers.

ent machines. Second, DARE needs to perform repair on network connections by propagating repair across machines. Third, DARE needs to minimize repair propagation during distributed repair. Finally, DARE needs to perform efficient repair on long-running server processes.

In the rest of this paper, §2 gives an overview of DARE, §3 describes how DARE solves the above challenges, §4 illustrates how DARE repairs a sample distributed attack, §5 and §6 present a preliminary implementation and results, §7 summarizes related work, §8 discusses open problems, and §9 concludes.

2 OVERVIEW

DARE consists of several components, shown in Figure 1. DARE's operation consists of three phases: normal execution of the distributed system, detection of an intrusion by an administrator, and distributed repair. The rest of this section gives an overview of Retro, and describes DARE's phases of operation in more detail.

2.1 Background

Retro [14] is an intrusion recovery system that repairs a single machine after a compromise by an adversary. It undoes the changes by the adversary, while preserving changes made by legitimate users. While the system is running normally, Retro records information about the system execution that it uses to build a detailed dependency graph of the system's execution, called an action history graph. The recorded information allows Retro to rollback the system state to a time in the past and selectively re-execute individual actions.

The action history graph consists of two types of objects: data objects, such as files, and actor objects, such as processes. Each object has a set of checkpoints, representing its state at various times. Each actor object additionally consists of a set of actions, representing the execution of that actor over some period of time. Each action has dependencies from and to other objects in the graph, representing the objects accessed and modified by that action.

Once an administrator detects an intrusion and identifies an *attack action*, Retro's repair controller uses the action history graph to perform repair by undoing the attack action as well as all of its effects. It does so by rolling back the direct effects of the attack action, skipping the attack action, and re-executing subsequent legitimate actions with dependencies to or from the objects affected by the attack. Retro separates the rollback-and-reexecute logic, implemented by the repair controller, from object- and action-specific logging, rollback, and re-execution, implemented by repair managers.

2.2 Normal execution

During normal execution, each machine running DARE constructs its own action history graph, by periodically saving file snapshots, and recording information about all system calls, including their arguments and return values. For system calls that operate on a network connection (e.g., `send` and `recv`), DARE records the connection's protocol and the source and destination IP addresses and ports. DARE models the connection as a socket data object that is present in the local action history graphs of both the machines at either end of the connection, and uses this recorded information about the connection to name that socket data object.

For example, Figure 2 shows action history graphs for a simplified version of the SourceForge attack. The `connect` and `send` system calls on the gateway machine, and the `accept` and `recv` system calls on the internal machine are modeled as actions writing to and reading from the same socket data object. This socket data object is present in both machines' action history graphs and connects them together. DARE inserts a single checkpoint on the socket data object at connection

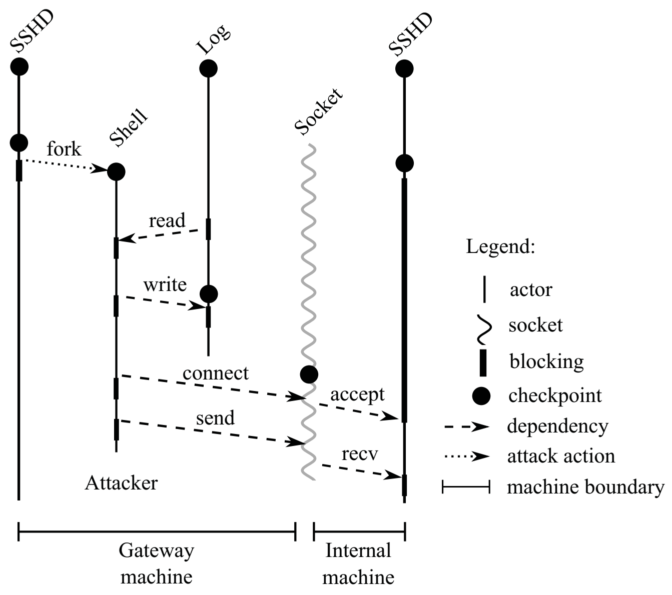

Figure 2: Action history graphs illustrating a simplified version of the SourceForge attack. The attacker uses a compromised account to log into `sshd` on the Internet-facing gateway, which spawns a shell. The shell reads and writes to the log that was used by administrator to later detect the attack. Finally, the attacker logs into an internal machine over SSH using the same compromised account. Both machines independently record their local action history graphs, which are connected by the socket data object.

start (i.e., before the `connect`), and the data object can be rolled back to this checkpoint during repair.

2.3 Repair

Each machine in the distributed system runs a DARE repair controller, which listens on a well-known port for repair requests from either the administrator or from a remote machine, performs the requested repairs locally, and initiates repair on remote machines as necessary.

Repair starts with an administrator detecting an intrusion and determining its source machine. DARE provides a GUI tool that an administrator can use to visualize the global action history graph across the entire distributed system. The administrator can use this tool to find the intrusion point from the attack symptoms, in a manner similar to BackTracker [15]. The administrator can also use other intrusion detection techniques such as retroactive auditing [17] and retroactive patching [8], to find the attack. The rest of this paper does not discuss intrusion detection further.

After finding the attack, the administrator invokes local repair on the source machine's controller by specifying the attack action. The controller reboots the machine to discard non-persistent state, and enters repair mode.

To undo the attack action, the controller first rolls back objects modified by the action to a checkpoint before the attack. The controller then replaces the attack action with a no-op in the action history graph, and uses the action history graph to determine other actions potentially affected by the attack. The controller re-executes each of those affected actions by first rolling back the action's inputs and outputs to the correct version, and then redoing the action. The controller repeats this process of determining affected actions and re-executing them until there are no more affected actions.

While a machine is undergoing repair, that machine is not available for normal execution. Though this can be inconvenient for users, it takes a potentially infected machine offline and helps contain further damage due to the attack spreading to other machines in the network.

When the controller on a machine has to redo a system call on a network connection, it propagates repair to the machine on the other end of the network connection, using the repair controller API described in §3.2.

As repair progresses, the controllers on all the machines to which the attack propagated are included in the repair process. Local repair happens simultaneously on these controllers, and they coordinate with each other before redoing system calls that operated on network connections. By the end of the repair process, DARE undoes all effects of the attack on the distributed system.

2.4 Assumptions

DARE makes several assumptions. First, we assume that attacks compromise only user-level services and that the attack does not tamper with the kernel, file system, or DARE's checkpoints and logs. Second, we assume that the administrator can identify all external attack actions. In particular, if an adversary steals a user's password or other credentials, our current design assumes an administrator pinpoints all improper uses of those credentials. Third, we assume that all the machines in the distributed system have DARE installed, and that the machines are under the same administrative domain, so that they can initiate repair on each other. We discuss how these assumptions may be relaxed in §8.

3 CHALLENGES ADDRESSED BY DARE

This section describes several challenges addressed by DARE's design in more detail.

3.1 Cross-machine dependencies

Propagating repair across machines in DARE requires addressing two problems: precisely identifying the network connection being repaired, and authenticating the repair request.

Identifying the network connection being repaired is complicated by the fact that source and destination ports can be reused over time. To uniquely identify connections, the DARE kernel module generates a random token for every connection it initiates, and includes it as a control option in the IP header of packets for that connection. When accepting a connection, the kernel module records the peer's random token and similarly includes it in all packets for that connection. During repair, when one controller sends a repair request to another controller, it includes the token in the request to identify the network connection to repair.

If adding IP options is undesirable, DARE kernel modules can communicate tokens for a network connection using an out-of-band channel. However, this incurs the overhead of an extra network round trip.

Authenticating repair requests between machines is important because an adversary may subvert an insecure repair mechanism to compromise the system. Our current design assumes that all of the machines are in the same administrative domain, and uses a secret cryptographic key, shared by all of the repair controllers, to authenticate repair requests, along with a nonce to ensure freshness.

3.2 Repairing network connections

During repair, the controller on a machine M_A may need to redo a system call A that operated on a network connection (e.g., `send`). This requires redo of the entire connection, and can result in repair on the machine M_B that is the other end of the connection.[2] To support this, each repair controller exports the following two API calls that can be invoked by another repair controller; these calls take the socket data object corresponding to the connection, identified by the connection's token, as the argument.

First, `rollback()` instructs the controller to roll back a specified socket data object to the single checkpoint before connection start, and start local redo on the actions that operated on that data object. `rollback` returns after redo is started on the first action that operated on the socket data object.

Second, a repair controller can send a `done()` message to the controller at the other end of the connection, to indicate that repair on its local socket data object is complete.

To redo the system call A on a network connection, M_A's controller invokes `rollback` on M_B's controller with the connection's token as the argument. M_B's controller initiates redo on the `accept` system call for that connection, and acknowledges M_A's `rollback` request. M_A's controller then re-executes the `connect` system call, establishing the connection. M_A and M_B's controllers continue redo of subsequent system calls on that connection, including A. Once repair is complete on the connection (e.g., at connection close), the controllers send `done` messages to each other.

3.3 Minimizing network replay

DARE borrows Retro's idea of *predicate checking*; Retro uses predicate checking to selectively re-execute only the actions whose dependencies are semantically different during repair. For example, if the attacker modified a file that was later read by a process P, Retro may be able to avoid re-executing P if the part of the file accessed by P is the same before and after repair.

DARE performs predicate checking on network connections to minimize distributed re-execution. For instance, in the SourceForge attack, the attacker's SSH client could have performed a DNS lookup during original execution. During repair, DARE can avoid re-executing the lookup on the DNS server if the DNS request was unchanged during repair.

One way to do predicate checking on a network connection is to compare the system calls issued on the connection during original execution with those issued during repair. However, this is insufficient, as non-determinism in the connection can cause the system calls to differ, even though their net effect is the same. For example, reduced network latency can cause one `recv` system call during repair to receive data that was originally received by multiple `recv` calls.

To solve this problem, DARE inserts a proxy *predicate checker* actor object in between the process using the network connection and the socket data object for the connection. The predicate checker compares the bytes sent by the process during repair with the bytes sent during original execution. As long as they match, the predicate checker replays the bytes received by the process during original execution back to the process, and repair is not initiated on the machine at the other end of this connection. When they do not match, the

[2] Assume that M_A is the client and M_B is the server for this network connection (i.e., M_A executed `connect` and M_B executed `accept` during connection setup.)

predicate check fails and the predicate checker initiates repair on the remote machine.

3.4 Repairing long-lived daemons

When repairing a process, DARE's repair controller re-executes it from the beginning. Although this works well for short-lived processes, many server processes in a distributed system, such as the SSH daemon, are long-lived daemon processes. If a daemon process was involved in an attack, repair would require re-executing the daemon process from its beginning (typically the boot time of the machine), which would be time-consuming.

One way to solve this problem is to periodically snapshot daemon processes using techniques developed for application migration and virtualization [1, 2, 4, 16]. This allows DARE's repair controller to roll back the daemons to a snapshot just before the attack. However, these snapshot mechanisms have significant runtime and storage overhead during normal execution. For example, a single snapshot of `sshd` using DMTCP [2] takes 0.6 sec and consumes 4 MB of disk space.

To avoid these overheads, we leverage the typical pattern of network services, which enter a *quiescent* state between servicing each request. For example, an SSH daemon has a known "connection accepting" state after servicing each request and spawning an SSH session. A daemon in the quiescent state is equivalent to having been restarted. Building on this intuition, DARE provides daemon developers with two options. First, the developer can have the daemon restart itself periodically, when it is in a quiescent state. This causes daemon processes to be short-lived, and allows DARE to repair only the particular daemon process that was the target of the attack, limiting the amount of re-execution.

Restarting the daemon can incur a performance penalty, and possibly lead to (brief) downtime. To address this limitation, a second option provided by DARE is a new `mark_quiescent` system call that the daemon developer can use to indicate that the daemon is in a quiescent state. During normal execution, DARE's logger records invocations of this system call. If the daemon needs to be repaired, DARE's repair controller restarts the daemon, but re-executes operations only from the last quiescent state before the action being repaired. Although this limits re-execution of operations before the attack, the repair controller must still re-execute subsequent operations on the affected daemon process, because an adversary could have subverted the process and the quiescent marks can no longer be trusted.

The quiescent marks can always be trusted in daemons that follow a common pattern of forking a child process to service each accepted connection; such daemons do not process any data from the network in the parent process, and thus cannot be compromised via the accepted network connections. For daemons that follow this pattern, the developer can additionally annotate the `mark_quiescent` system call as trusted, indicating to DARE that it can skip re-executing operations on the daemon process following the next quiescent mark after the action being repaired.

4 REPAIRING THE SOURCEFORGE ATTACK

To understand how DARE repairs from an intrusion in a distributed system, consider a simplified version of the SourceForge attack, where an attacker logs into the gateway machine using a compromised user account, and from there proceeds to log into an internal machine. Figure 2 shows the action history graph for this attack. Once the administrator identifies the intrusion and pinpoints the attacker's entry point on the gateway machine, DARE's repair controller on the gateway machine rolls back `sshd` and removes the attacker's login action from the action history graph. This causes the shell to not be forked, and the `connect` to internal machine's `sshd` and the subsequent `send` to not be invoked. The gateway's controller proceeds to undo the `connect` and the `send` system calls. It rolls back its local socket data object, and invokes `rollback` on the internal machine's controller, which rolls back its own local socket data object. The gateway's controller undoes the `connect` and `send` system calls, and sends a `done` message to the internal machine's controller indicating that repair on the socket data object is done. The internal machine's controller then undoes the `accept` and `recv` system calls, which subsequently leads to undoing the attacker's `ssh` session and all causal effects thereof. The repair controllers continue the repair process until all effects of the attack are undone on all the machines.

5 IMPLEMENTATION

We implemented an early prototype of DARE for Linux, building on top of Retro. Figure 3 shows the number of lines of code for different DARE components. The DARE kernel module interposes on all system calls by remapping the `syscall` table in the kernel, collects system call information needed for dependency tracking, and sends them to the user-level DARE daemon via `relay-fs`. The kernel module also implements the new

Component	Lines of code
Logging kernel module	3,300 lines of C
AHG GUI tool	2,000 lines of Python
Repair controller, managers	5,300 lines of Python
System library managers	800 lines of C

Figure 3: Lines of code of different components of the DARE prototype.

`mark_quiescent` system call by overwriting an unused system call in the `syscall` table. We implemented a GUI tool that, given the recorded DARE logs from different machines, displays a global action history graph by connecting local graphs from different machines.

6 EVALUATION

To evaluate our preliminary DARE design, we wanted to show that it can recover from a distributed attack, and that the techniques described in §3 reduce the amount of re-execution.

For evaluation, we constructed a simplified version of the SourceForge attack. The experimental setup consists of two machines running Linux with DARE installed, corresponding to the gateway and internal machines in the SourceForge attack. The internal machine runs a modified version of `sshd` that forks a separate process to handle each accepted connection, and calls `mark_quiescent` after it launches each SSH session. The `mark_quiescent` is annotated as trusted.

Our test workload consists of 5 legitimate users using `ssh` to log into the internal machine from the gateway machine, followed by an attacker logging into the gateway machine and from there into the internal machine, followed by another 5 legitimate user logins. Each SSH session writes session information to an append-only log file. This workload generates a total of 8,953 nodes in the action history graphs of both the machines.

In our test workload, the administrator identifies the attack by inspecting the action history graph using the GUI tool, and initiates repair by identifying the attacker's login on the gateway machine. The repair controller on the gateway machine initiates remote repair on the internal machine.

We consider two scenarios. In the first scenario, `sshd` lacks quiescent marking, in which case the internal machine's repair controller restarts `sshd` from the beginning, re-executes all the 10 legitimate user logins and skips the attacker's SSH session. In the second scenario, `sshd` has quiescent marking, as described above. In this case, the repair controller restarts `sshd` and re-executes starting from the quiescent period before the attacker's SSH session: it skips the attacker's session, and reruns the writes to the log file by the subsequent 5 legitimate user sessions. Repair with and without quiescent marking take 0.44 seconds and 3.7 seconds, respectively, showing that quiescent marking works well in practice.

7 RELATED WORK

The two closest pieces of work related to DARE are the Retro [14] and Warp [8] intrusion recovery systems, which provide intrusion recovery for a single machine and for web applications, respectively. DARE builds on Retro's rollback and re-execute approach to provide intrusion recovery for a distributed system.

Existing intrusion detection [6, 15] and intrusion auditing systems [17] allow an administrator to detect compromises in a distributed system. The administrator can use them in conjunction with DARE, and can use DARE to recover from an attack once it has been identified by one of these systems.

Past work on worm containment [11] limits the number of machines infected by an attack by automatically detecting the attack, generating worm filters from infected machines, and deploying the filters on uninfected machines to prevent spread of the worm. However, the infected machines still need to be repaired after the attack. DARE can perform this repair and is thus complementary to worm containment systems.

8 OPEN PROBLEMS

DARE does not currently track the re-use of stolen credentials by an adversary. This works fine for authentication schemes that are resilient to replay attacks, such as authentication using a remote SSH agent or the increasingly popular one-time password schemes like RSA SecurID [5] and Google Authenticator [3]. However, traditional password authentication schemes are still prone to replay attacks. For example, an adversary could break into the system, steal a user's password, and later use that password to log into the system again. Such attacks are difficult for DARE to handle because they involve machines outside of DARE's control. For credentials that are easy to identify, such as SSH private keys or Kerberos tickets, DARE could track access to and use of credentials, and determine suspect uses of credentials that were accessed during an attack. Passwords are harder to identify, and would likely require the administrator to help identify accounts whose passwords have been compromised. Once stolen credentials are identified, DARE can identify and undo suspect logins.

DARE assumes that all of the machines in the distributed system are under the same administrative domain. Extending DARE to repair between mutually distrustful machines would require several changes. First, the tokens exchanged during normal execution of network system calls need to be cryptographically secure, so that they cannot be forged in remote repair requests. Second, each administrative domain needs to have a policy in place indicating what remote repair requests are allowed. Finally, the two-phase rollback-and-reexecute model may need to be replaced with a model where one machine sends another machine a complete proposed change to some past message.

DARE's repair controller API handles repairs that cancel a network connection or modify the data that was sent over the connection. Adding new network connections during repair requires naming the new connections and adding them to the right point in the action history graph's timeline, which DARE does not currently support.

9 CONCLUSIONS

DARE helps system administrators recover system integrity after an attack that spreads between several machines in a cluster. DARE uses Retro's rollback-and-reexecute approach to recover individual machines. Across machines, DARE tracks dependencies by assigning unique tokens to network connections. DARE propagates changes between repair controllers on each machine during repair, and minimizes distributed re-execution with predicate checking on network connections. Finally, DARE allows software developers to annotate quiescent periods in their code to reduce re-execution of long-lived processes. An initial prototype of DARE can repair from a simplified version of the SourceForge attack, and the above techniques reduce the amount of re-execution necessary during repair.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This research was partially supported by the DARPA CRASH program (#N66001-10-2-4089), and by NSF award CNS-1053143.

REFERENCES

[1] CryoPID - A Process Freezer for Linux. URL `http://cryopid.berlios.de/`.

[2] DMTCP: Distributed MultiThreaded CheckPointing. URL `http://dmtcp.sourceforge.net/`.

[3] Google Authenticator - Two-step verfication. URL `http://code.google.com/p/google-authenticator/`.

[4] OpenVZ Wiki Main Page. URL `http://wiki.openvz.org/Main_Page`.

[5] RSA SecurID - Two-Factor Authentication, Security Token. URL `http://www.emc.com/security/rsa-securid.htm`.

[6] The snort intrusion detection system. URL `http://www.snort.org`.

[7] kernel.org compromised, January 2011. URL `http://lwn.net/Articles/457142/`.

[8] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 101–114, Cascais, Portugal, October 2011.

[9] Community Team. Sourceforge Attack: Full Report, January 2011. URL `http://sourceforge.net/blog/sourceforge-attack-full-report`.

[10] Community Team. SourceForge.net passwords reset, January 2011. URL `http://sourceforge.net/blog/sourceforge-net-global-password-reset`.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–147, Brighton, UK, October 2005.

[12] J. Dunagan, A. X. Zheng, and D. R. Simon. Heat-ray: Combating identity snowball attacks using machine learning, combinatorial optimization and attack graphs. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[13] N. Falliere, Murchu, and E. Chien. W32.Stuxnet Dossier. Symantec Security Response online report, February 2011.

[14] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proc. of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

[15] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, February 2005.

[16] O. Laadan and S. E. Hallyn. Linux-CR: Transparent application checkpoint-restart in Linux. In *Proc. of the 12th Annual Linux Symposium*, Ottawa, Canada, July 2010.

[17] X. Wang, N. Zeldovich, and M. F. Kaashoek. Retroactive auditing. In *Proc. of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, July 2011.