WormHealer: Replay-based Full-System Recovery from Control-Flow Hijacking Attacks

Daniela A. S. de Oliveira   Jedidiah R. Crandall   Gary Wassermann
S. Felix Wu   Zhendong Su   Frederic T. Chong
University of California, {Davis, Santa Barbara}
{oliveira,crandall,wassermg,wu,su}@cs.ucdavis.edu chong@cs.ucsb.edu

Abstract

System availability is difficult for systems to maintain in the face of Internet worms. Large systems have vulnerabilities, and if a system attempts to continue operation after an attack, it may not behave properly. Traditional mechanisms for detecting attacks disrupt service and can convert such attacks into denial-of-service. Current recovery approaches have at least one of the following limitations: they cannot recover the complete system state, they cannot recover from zero-day exploits, they undo the effects of the attack speculatively or they require the application's source code be available. This paper presents WormHealer, a replay-based, architecture-level post-attack recovery framework using VM technology. After a control-flow hijacking attack has been detected, we replay the checkpointed run using symbolic execution to discover the source of the malicious attack. We then replay the run a second time but ignore inputs from the malicious source. We evaluated WormHealer on five exploits for Linux and Windows. In all cases, it recovered the full system state and resumed execution. It also recovered all TCP connections with non-malicious clients and the communication that had been taken place during the attack, except for some limited cases.

1 Introduction

System availability is difficult for online systems to maintain in the face of Internet worms. Control-flow hijacking Internet worms perform their attacks by overwriting control data in a victim host, which allows them to perform arbitrary malicious actions. A system without vulnerabilities (or only patched vulnerabilities) is not at risk, but large, real-world systems do have vulnerabilities. Indeed, the Recovery Oriented Computing (ROC) paradigm views faults, bugs, and errors not as problems to be solved, but as facts to be coped with [23]. Intrusion detection systems (IDSs) may be able to detect and stop an attack, but they typically disrupt service for all users. For example, if an IDS reboots the system, the system will drop non-malicious clients, lose recently acquired data, and experience down-time as it restarts and replenishes cached data for services like DNS. Alternative actions, such as trapping to the OS, do not offer better solutions for detangling the malicious behavior from the non-malicious, and they suffer from the same problems as those associated with rebooting, although in varying degrees. Even if the system continues its execution after the attack (by killing the offending application or process, for instance), it could not be able to proceed. The exploit could have corrupted areas of memory used by non-malicious processes and the system could eventually crash or hang. Also, the attacker could have damaged critical system files or data structures and the continuation of the host execution could lead to a situation of error or inconsistency. A malicious host that repeatedly sends attacks can aggravate these problems by forcing repeated reboots (or other actions) and effectively performing a denial-of-service attack.

The research literature proposes some techniques to address this problem, but each technique has at least one of the following limitations: it cannot recover or repair damage to the file system [26–28]; it loses state related to interprocess communication, messages, signals, and resources (*e.g.,* files opened, processes spawned, pages allocated, etc.) [14, 26]; or it cannot recover from zero-day exploits or it undoes the effects

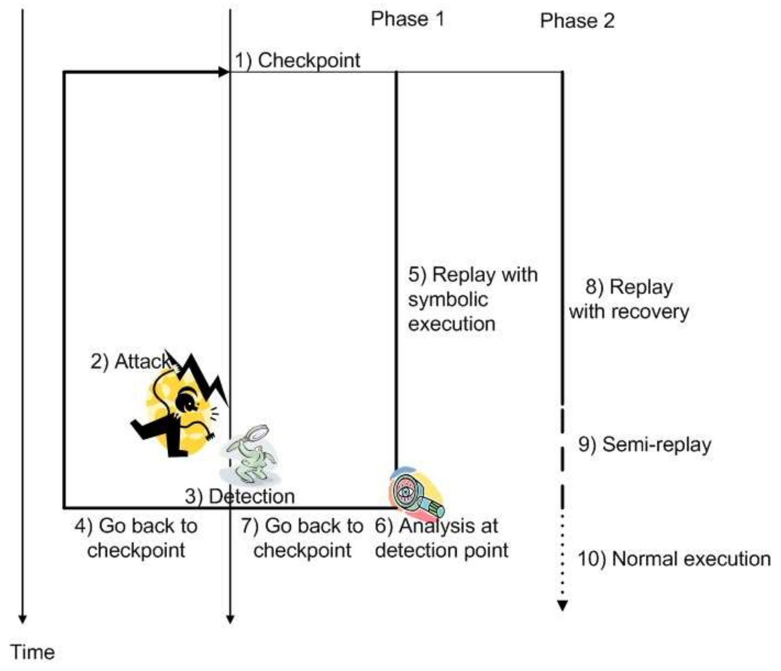

Figure 1: **WormHealer**

of the attack speculatively [25, 27]. There are also approaches that perform recovery only on the level of an application, without considering the complete system state of the victim host [25, 27, 28]. These require that the application's source code be available.

Automated recovery from worm attacks is challenging because it requires (i) detecting that a host is under attack and stopping the attack, (ii) analyzing the attack to discover its source, (iii) distinguishing the actions of the exploit, and (iv) reverting the attack actions without side-effects. This paper presents an OS- and application-independent approach to recover from zero-day control-flow hijacking attacks using VM technology. It both enables the system to proceed without losing any state and prevents future attacks from the malicious source. Our approach is based on analyzing an attack once it is discovered to identify its source, and using a log and replay framework to go back and re-execute without the malicious data. We have implemented our approach as WormHealer.

Figure 1 provides an overview of our recovery strategy. A host executing on a WormHealer-enabled VM starts its execution logging all architecturally non-deterministic activity (*e.g.*, interrupts and input events). In the logging process, we record information about where each input event from the network came from (*i.e.*, IP address and source port). The VM allows that frequent light-weight checkpoints (based on copy-on-write) of the complete system state (including the file system) be taken.

In the event of an attack, the VM goes back to the most recent checkpoint and replays the system in symbolic execution mode. In this mode, the VM tracks all bytes that come in from the network as they propagate through the system. When the replay reaches the point where the detector stops the attack, WormHealer discovers the attack's source. The system then returns to the checkpoint and replays execution up to the point where the VM sees the first malicious input entering the system. The VM then ignores this malicious event and enters a *semi-replay* phase. In this phase the VM is executing partially in normal-mode and in replay mode: it accepts new events, but also replays non-malicious network packets so that they can continue as though no attack had happened. We perform semi-replay because after removing the malicious events from the run we cannot continue pure deterministic replay. Some non-malicious events in the log file may be dependent or consequence of the malicious events we have taken away from the run. When the last non-malicious network packet from the log file is processed, the system returns to its normal mode. However, from this point on, all network packets coming from the detected malicious source are ignored by the VM, thus preventing the host from receiving other instances of the same attack from the same source.

WormHealer was designed using three components as building blocks: a detection system, a symbolic execution tool and a replayer. All these components, together with WormHealer itself, run as extensions to a host VM [29]. The detection system is able to catch zero-day control-flow hijacking attacks. The symbolic execution tool can track down bytes coming from network packets during its lifetime in the system and provides WormHealer with data so that it can discover the source of an attack. The replayer will allow WormHealer to undo the effects of the exploit as if it could go back in time and change the past (*i.e.*, the fact that the host has been attacked).

The contributions of our work are as follows:

- we propose an architecture-level strategy for recovering from all control-flow hijacking attacks detected by our IDS that does not lose system state or disrupt service;
- we propose a technique to distinguish malicious and non-malicious architecture-level inputs in the system. This information is useful for forensic analysis, which asks how the attack propagates through the system, what its effects are, and what the vulnerability is that it exploits;
- we propose an automated technique to prevent repeated attacks from a malicious source;
- we have implemented our approach and we present experimental results from it. In particular, we show that it is possible both to identify which network bytes are responsible for an attack and to continue service to non-malicious clients (even if the non-malicious client's connection thread are in the same process address space that had been corrupted by the attack).

The rest of the paper is organized as follows. Section 2 describes in detail WormHealer's design and implementation. In Section 3 we present the experiments we have performed to validate our recovery approach. Section 4 discusses related work. Our conclusions and future work are presented in Section 5.

2 WormHealer Design and Implementation

2.1 High-level View of Worm Healer

WormHealer is an architecture-level recovery framework running as an extension to the Bochs VM [34] whose main components are a detection system, a symbolic execution tool and a full-system replayer. It can recover the complete state of a system after a control-flow hijacking attack detected by our detection system and works as follows: in the event of an attack, it goes back to the most recent checkpoint and replays the system in symbolic execution mode. In this mode it tracks all bytes that come from the network during that run. When the detection system stops the attack during replay, WormHealer discovers its source. Then, it causes the VM to go back to that previous checkpoint again for recovery through replay (during these replays WormHealer does not re-send network packets). The complete system execution is replayed until the first malicious input enters the system. The VM then do not process this malicious event (and all subsequent events coming from the same source) and enters a semi-replay phase.

In this phase the VM executes partially in normal mode (without logging) and replay mode: it accepts and process new events, but also replays non-malicious network packets so that their effects can be propagated up to the point the attack was originally detected. When the last non-malicious network packet is replayed, the system returns to its normal mode. However, from this point on, all network packets coming from the detected malicious source are ignored, thus, preventing the host from falling victim to repeated attacks from the same source. Figure 1 provides an overview of WormHealer recovery approach.

2.2 Building Blocks of WormHealer

As we have mentioned, WormHealer is designed upon three main components: an attack detection system, a symbolic execution tool and a replayer. The next subsections describe each one of these building blocks.

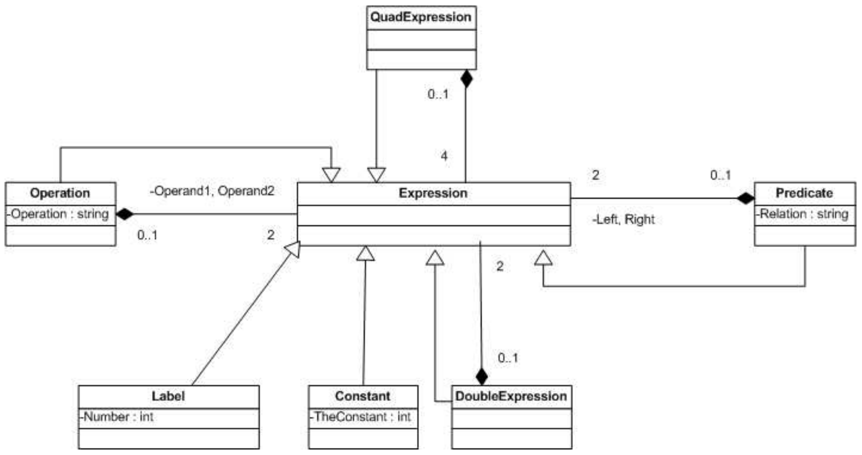

Figure 2: **DACODA Object Hierarchy**

2.2.1 Attack Detection

The first requirement of any post-attack recovery approach is to detect that a host is under attack. WormHealer uses the Minos [9] microarchitecture to accomplish this. Minos is currently implemented as an extension to the Bochs VM and catches zero-day control-flow hijacking attacks. It implements Biba's low-water-mark integrity policy [4] on individual words of data. Minos can detect attacks that corrupt control data to hijack program's control flow, where control data is considered any data that is loaded into the program counter (EIP) on a control flow transfer instruction or any data used to calculate it, for instance, return pointers, function pointers, jump targets, etc...

In Minos, every 32-bit word of memory and general-purpose register is augmented with an integrity bit, which is set by the kernel when it writes data into them. This bit is set to low or high, depending on the trust the kernel has for it. Data coming from the network are usually regarded as low integrity. Any control transfer involving untrusted data is considered a vulnerability, and a hardware exception traps to the VM whenever this occurs.

WormHealer current prototype only recovers from control-flow hijacking attacks detected by Minos.

2.2.2 Attack Analysis

The second and third requirements of a post-attack recovery approach are to discover the source of the attack and distinguish what the actions of the exploit in the victim host. We have extended the DACODA symbolic execution tool [10] to meet this requirement. DACODA is an extension of a Minos-enabled Bochs VM and can track down network bytes by performing full-system symbolic execution [19] on every machine instruction.

DACODA makes use of a symbolic memory space (including the memory of the Ethernet device) and a symbolic register bank to store information about the propagation of network bytes in our system. Each component of this symbolic storage area has a 1:1 correspondence with the real component in the system architecture. The network bytes propagation information is stored in objects called *Expressions*, which can be of several types, for instance, Label, Operation, Predicate, DoubleExpression and QuadExpression. Initially the symbolic storage area has no Expression associated with any of its locations. Figure 2 illustrates DACODA object hierarchy in UML notation.

A Label has a unique integer that identifies it. An Operation is characterized by the operation that is performed in its two operands, which can be any type of Expression. A Predicate is formed by a specific relation comparing an Expression on the left-hand side and another on the right-hand side. A DoubleExpression represents the two Expressions associated with each byte of a 16-bit word, and a QuadExpression is formed by the four Expressions associated with the four bytes of a 32-bit word.

```
1. mov     al,[AddressWithLabel1832]
; AL.expr <− (Label 1832)

2. add     al,4
; AL.expr <− (ADD AL.expr 4)
; AL.expr == (ADD (LABEL 1832) 4)

3. cmp     al,10
; ZF.left <− AL.expr
; ZF.left == (ADD (Label 1832) 4)
; ZF.right <− 10

4. je  JumpTargetIfEqualToTen
; P = Predicate(EQUAL ZF.left ZF.right)
; P == (EQUAL (ADD (Label 1832) 4) 10)
; if (ZF == 1)
;   AddToSetOfKnownPredicates(P);
; Discover predicate if branch taken
```

Figure 3: **DACODA Example**

We describe how DACODA works with the example given in Figure 3. First, when a network packet is read from the Ethernet device, we create a Label object identified by a unique integer for every byte of the packet. Suppose a byte from a network packet is labeled with "`Label 1832`" when it is read from the Ethernet card. As the byte is stored in the Ethernet device memory, its corresponding label object is stored in the Ethernet device symbolic memory. DACODA then follows the byte through the device into the processor where the kernel reads it into a buffer. In this case wherever the byte is stored, DACODA tracks it down through its Label in the symbolic storage space. Suppose the kernel copies this byte into user space and a user process moves it into the AL register, adds the integer 4 to it, and makes a control flow transfer predicated on the result being equal to 10.

For the first instruction DACODA stores an Expression of type Label (number 1832) into our symbolic AL register. For the second instruction, DACODA creates an Operation object with operation **ADD**, operand1 **Label 1832** and operand2 **Constant 4** and stores this object into our symbolic AL register. The Zero Flag (ZF) is used by the Pentium for indicating equality or inequality. DACODA associates two expressions with ZF, left and right, to store the expressions for the last two data that were compared. ZF can also be set by various arithmetic instructions but only explicit comparison instructions set the left and right pointers in DACODA. Thus, in the third instruction DACODA sets the left expression of ZF to be the expression stored at the AL symbolic register and the right expression to **Constant 10**. If any subsequent instruction checks ZF and finds it to be set, DACODA creates an equality predicate. This is exactly what happens for the last instruction in our example.

2.2.3 Execution Log and Replay

The fourth requirement of a post-attack recovery system is to revert the attack actions in the victim host. We achieve this through replay. The replay capability allows us to go back in time and undo the effects of the attacker in the system. WormHealer extends the ExecRecorder [12] log and replay framework to meet this requirement. ExecRecorder can replay the execution of an entire system by checkpointing the complete system state (virtual memory and CPU registers, virtual hard disk and memory of all virtual external devices) and logging all architectural nondeterministic events (interrupts and input events).

The checkpoint strategy involves the complete system state but is light-weight because it is based on copy-on-write. It is implemented by duplicating the VM process via the *fork* system call. After the fork, the parent process waits in the background for a SIGUSR1 signal while the child process continues its

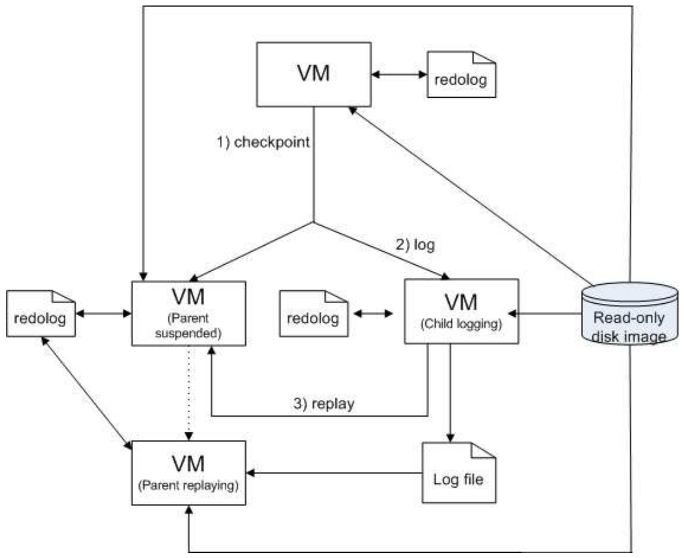

Figure 4: **ExecRecorder**

execution logging architectural nondeterministic events. The suspended parent represents the frozen state of the system at the time the checkpoint was taken. The checkpoint of the virtual HD is achieved by using the Bochs undoable disk mode. An undoable disk is a committable/rollbackable disk image based on a read-only disk image combined with a file, called *redolog*, that contains all changes made to the read-only image. After a run the *redolog* file can be merged to the read-only image or simply discarded. ExecRecorder always starts the VM with the read-only disk image. When a checkpoint is taken, the child process continues its execution with a new *redolog* file, which is initialized with the contents of the parent process *redolog* file.

After the logging of a run the replay module can be called to reproduce the system execution from a certain checkpoint. The child process wakes up the parent process with a SIGUSR1 signal. The parent process, which captures the system state at the point the checkpoint was taken, resumes its execution. The virtual disk image used is also the one at the time the checkpoint was taken. However, all interrupts or input events that may be generated are disabled and do not affect the state of the system. Being in replay mode, the VM uses all information recorded in the log file to reproduce the events at the tick at which they happened during the log phase. Figure 4 illustrates how ExecRecorder works.

2.3 Design and Implementation

We have used Bochs [34] as our virtual machine. It is a host VM [29] that emulates the IA-32 Pentium architecture. The host OS in our architecture is Linux 2.6. WormHealer, along with the detection microarchitecture (Minos), the symbolic execution tool (DACODA) and the log and replay framework (ExecRecorder), run as an extensions to Bochs. We have designed this system to run as a honeypot. Figure 5 illustrates this architecture.

2.3.1 Identifying Network Inputs with their Sources

When WormHealer is enabled, its main assumption is that the system can be attacked anytime and it should start acting before this happens. Therefore, our VM always starts its execution in log mode. The logging mode allows WormHealer to be able to go back in time (*i.e,* to a previous checkpoint) in the event of an attack. Frequent light-weight system checkpoints can be taken using ExecRecorder, as explained in section 2.2.3. The recovery time is directly proportional to the length of the logged run and, as a result, frequent checkpoints contribute to low recovery times.

WormHealer associates a network input event with its source as follows. When an Ethernet frame is

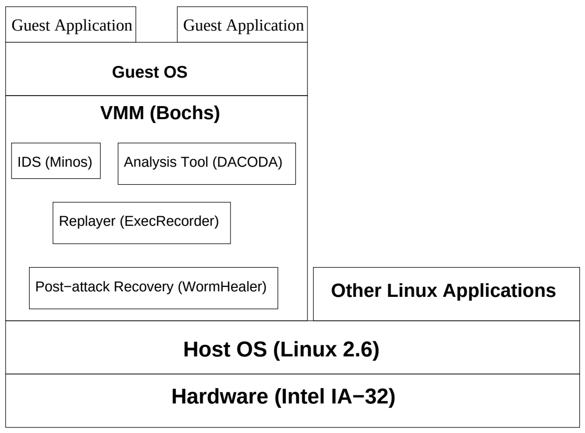

Figure 5: **Our VM architecture**

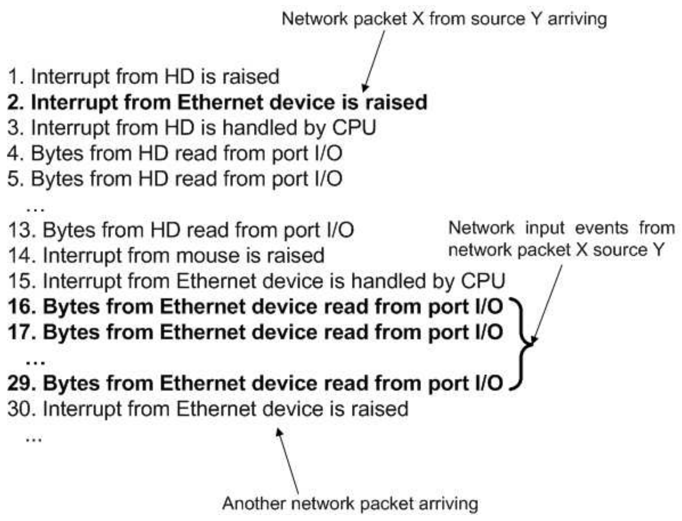
Network packet X from source Y arriving
1. Interrupt from HD is raised
2. Interrupt from Ethernet device is raised
3. Interrupt from HD is handled by CPU
4. Bytes from HD read from port I/O
5. Bytes from HD read from port I/O
...
13. Bytes from HD read from port I/O
14. Interrupt from mouse is raised
15. Interrupt from Ethernet device is handled by CPU
16. Bytes from Ethernet device read from port I/O
17. Bytes from Ethernet device read from port I/O
...
29. Bytes from Ethernet device read from port I/O
30. Interrupt from Ethernet device is raised
...
Network input events from network packet X source Y
Another network packet arriving

Figure 6: **Identifying the source of network input events**

read from the Ethernet device, we extract information about the source of this packet at the Network and Transport protocol levels. WormHealer maintains this information (source IP address and port, destination IP address and port) in an object that we call **networkSource**. Each networkSource is uniquely identified by an id, that we call **networkSourceId**.

When WormHealer extracts the networkSource of an Ethernet frame it keeps its id as the current in the system until a new packet from another networkSource arrives. After finishing reading the frame, the Ethernet device will interrupt the CPU, signaling that there are bytes to be read through port I/O. This interrupt may not be handled immediately by the CPU because it may have to handle higher-priority interrupts, for instance. When this interrupt is actually handled by the CPU, the corresponding network bytes are read through port I/O. Thus, all network input events occurring until the next Ethernet frame arrives have as its networkSourceId the one kept as current by WormHealer. These events, together with interrupts, are logged by ExecRecorder and WormHealer extends the latter to also log, network packets and, only for network input events, its original networkSourceId. Figure 6 shows an example of this sequence of events.

2.3.2 Identifying Malicious Network Inputs

If Minos catches an attack during the run, instead of letting a hardware exception trap to the VM, WormHealer invokes ExecRecorder to replay the logged run in symbolic execution mode. During this replay (which does not re-transmit network packets) DACODA gathers information about how the network bytes propagate into our system.

As explained in section 2.2.2, for each byte in a network packet read from the Ethernet device, DACODA creates a Label object with an unique id in the system. We extended DACODA's Label definition so that it can also store information about the source of this byte. Then, when a Label is created for each of the frame's bytes read by the Ethernet device, we also store in that Label the id of the byte's network source.

The attack will be inevitably replayed but, at this time, at the detection point, WormHealer will inspect the symbolic storage area (symbolic memory and registers) to discover the source of the attack. This symbolic storage area stores Expressions representing the symbolic execution of that run. More specifically, WormHealer will inspect in the symbolic storage area the Expressions located at the same memory address or register from which the bogus value for the register EIP came from. This memory address or register was corrupted at same point by the exploit and the value at this location is derived from the malicious bytes the attacker sent through a network packet.

WormHealer does something that is extremely challenging for signature-generation approaches: it associates the raw bytes from the network packet directly related to the attack with the bogus EIP value. Current signature schemes cannot do this because of metamorphism. In metamorphism the malware body may appear encrypted in the network packet and is decrypted by the exploit mechanism itself during the attack. In spite of any metamorphic technique, the bogus EIP value is derived from certain bytes that came in one or more network packets from the malicious source. We can trace this value back to the bytes of the packet if we know how these bytes propagated into the system during their lifetime. We have accomplished this through full-system symbolic execution.

From the set of Expressions inspected by WormHealer, we extract all associated Labels, which correspond to network bytes. Then, we inspect the Labels corresponding networkSourceId. All Labels will have the same networkSourceId (as found by WormHealer), which is the identification number of the malicious network source. Next, WormHealer initiates the recovery phase through replay.

2.3.3 Replaying to Recover from Attacks

WormHealer starts replaying the run knowing what is the id of the malicious network source. During this replay (which does not re-send network packets) for each network input event processed, WormHealer compares the logged networkSourceId for the event with the malicious networkSourceId. If they are different the event is replayed because it is non-malicious. If they are the same, this means that WormHealer is seeing the first malicious input event entering the system. The malicious event is not processed and the VM enters a phase that we call *semi-replay*.

This phase represents our solution to the problem of the *time-travel paradox* discussed in [6]. Brown and Patterson compare their 3R's model (rewind, repair and replay) to the metaphor of time-traveling that we have seen portrayed in a very famous movie. As described in [6], the protagonist goes back in time to redo or repair some action that caused some bad consequences to the present. After making the necessary changes, the protagonist inevitably affects the course of other actions and when he returns to the present, he sees a "new" or "modified" present that is consequence of the bad action he repaired along with the inevitably side-effects on other aspects of the past.

In our particular instance of time-travel, the repair action is not process a malicious input as if it had never happened. Up to the point that we see the first malicious input entering the system we can replay the run without worrying about side-effects. When we see the first malicious input entering the system we cannot just ignore (not process) all malicious inputs while replaying other events because certain non-malicious events may be dependent of some malicious events and we could reach a situation of error or inconsistency.

In our solution, when WormHealer sees the first malicious network input entering the system, we enter

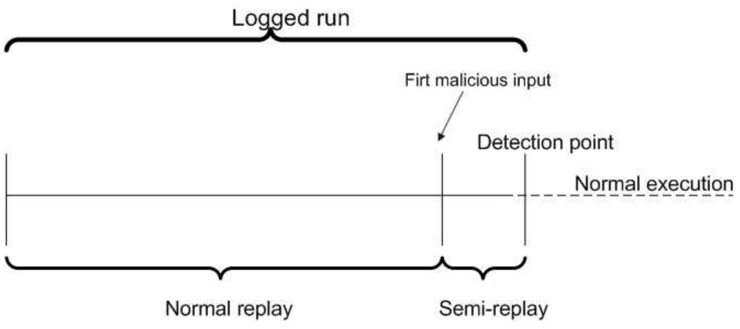

Figure 7: **The Semi-Replay Phase**

Exploit	**OS**	**Port(s)**	**Class**	**bid [35]**	**Vuln. Discovery**
DCOM RPC (Blaster)	Windows XP	135 TCP	Buffer Overflow	8205	Last Stage of Delirium
IIS (Code Red II)	Windows Whistler	80 TCP	Buffer Overflow	2880	eEye
wu-ftpd Format String	RedHat Linux 6.2	21 TCP	Format String	1387	tf8
rpc.statd (Ramen)	RedHat Linux 6.2	111 & 918 TCP	Format String	1480	Daniel Jacobiwitz
innd	RedHat Linux 6.2	119 TCP	Buffer Overflow	1316	Michael Zalewski

Table 1: Worm exploits

a semi-replay phase. As the name suggests in this phase the VM is executing partially in replay mode and in normal mode. The VM re-enables interrupts and accepts new input events (other than those prescribed in the log file). However, it rejects all malicious network packets. This strategy prevents the host from falling victim to repeated attacks from the same malicious source. The VM keeps replaying non-malicious network packets from the log file until all of these events are consumed. Our goal with this strategy is bifold: replay the system as if the malicious inputs had never occurred and bring the system state (memory, file system, process, etc...) to a legitimate state: a state it would have reached had the attack never happened. When all non-malicious network packets in the log file are consumed the system enters normal mode but continues to refuse any network packet coming from the malicious source. It is important to point out that the system entropy after recovery will be different from what it was at the time the exploit was stopped. This is because during recovery, we remove from the system all malicious events, thus affecting the values at the system entropy pool. Figure 10 illustrates our semi-replay phase.

3 Experimental Evaluation of WormHealer

The main goal of our experiments was to verify if WormHealer could effectively recover a system after a control-flow hijacking attack detected by Minos. More specifically we analyzed if WormHealer could (i) discover the source of the attack and distinguish malicious and non-malicious events (ii) recover the server execution along with its complete state (file system, resources opened, interprocess communication, etc...) (iii) recover the communication of the server with remote non-malicious clients during the semi-replay phase.

3.1 Experimental Setup

We have selected 5 well-known exploits to analyze how effectively WormHealer could provide recovery. These exploits represent buffer overflow or format string vulnerabilities from three different operating systems: Linux 2.4.21, Windows XP and Windows Whistler. Table 1 describes the selected exploits.

First we have analyzed if a WormHealer-enabled server could continue its execution after being attacked by each one of these exploits. For each exploit we have analyzed the server in three situations: idle, executing intensively its CPU, disk and running multitask activities, and running a Web server in a noisy campus network. For Linux, we have selected the UnixBench [32] as our CPU/disk/multitask activities benchmark. It is a benchmark suite for Linux that integrates CPU, file I/O, process spawning and other workloads. The following tests were performed: Dhrystone 2 using register variables, arithmetic, system call overhead, pipe throughput, pipe-based context switching, process creation, execl throughput, file system throughput, concurrent shell scripts, compiler throughput, and recursion. For Windows we have selected an implementation of the Sieve of Erastosthenes. Our goal was to generate a CPU-intensive workload. We have chosen to use this algorithm not only because it is usually part of several well-known CPU benchmarks, but also because publicly available CPU benchmarks for Windows were interactive and we wanted automated microbenchmarks. The Web server running in our Linux guest OS is Apache 1.3 and in Windows XP and Whistler is Apache 2.0.

Second, we have analyzed how our recovery approach affected a Web client communicating with a WormHealer-enabled Web server. We have simulated the load of Web clients with the Webstone 2.5 benchmark [33]. The Web client also executed in a Bochs VM running Linux 2.4.21. As our Web server was running on a VM making use of basic interpretation we have decided to simulate our clients in the same type of VM. All the experiments were executed on a Pentium 4 SMP with 2 3.2 GHz CPU's and 1 GB of RAM. Also, each experiment was executed three times and its results averaged.

3.2 Effectiveness of the Recovery

3.2.1 Can Server Continue After Recovery From Attacks?

Here we analyze if a server could continue its execution after being attacked by each one of these exploits. As we have mentioned in the last section, for each exploit we have analyzed the server in three situations: idle, executing intensively its CPU, disk or running multitask activities, and running a Web server.

We have attacked our WormHealer-enabled server with all the exploits presented in Table 1 and, for all of them, the server continued its execution without losing its full-system state. The microbenchmarks finalized successfully without any side-effects. In our experiments as soon as the microbenchmark started executing we launched the attack.

Figures 8 through 12 show, for each exploit, the number of instructions executed from the time the malicious input entered the system through the network (attack injection) up to the time the control-flow hijacking attempt was detected.

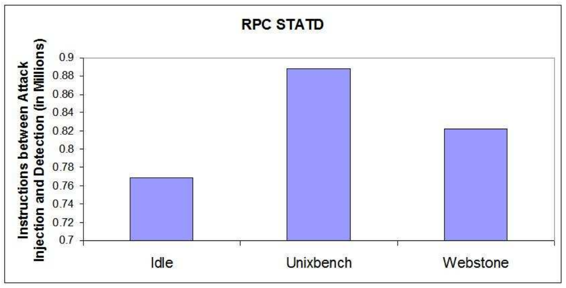

Figure 8: rpc.statd (Ramen)

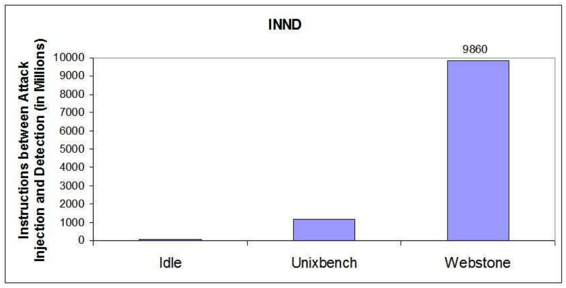

Figure 9: innd

The number of instructions executed between attack injection and attack detection determine the length of the semi-replay phase. The smaller the semi-replay phase the less our recovery approach can cause side-effects or affect a client communicating with the server. As we have mentioned before, during semi-replay the system entropy is different from what it were during normal mode and under attack. This is because during semi-replay we remove all malicious events from the execution, thus changing the values in the system entropy pool. If during semi-replay the communication between client and server depends on newly generated random numbers, for instance, an authentication key or a TCP initial sequence number, the numbers chosen during semi-replay will be different from those chosen when the system was under attack. In this situation the remote client will keep trying to communicate with the server with the old values (*i.e.*,

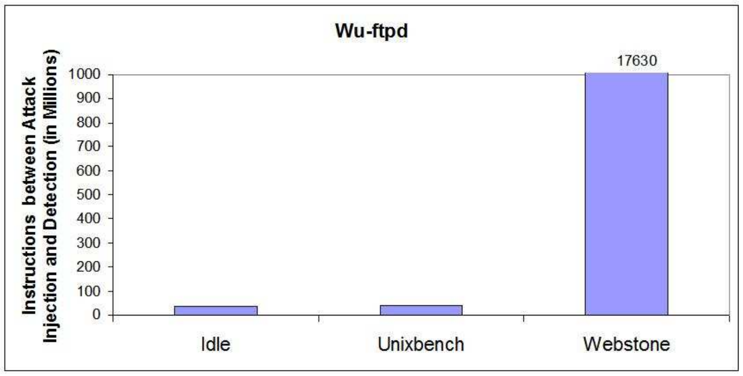

Figure 10: wu-ftpd

the values the server chose during normal mode and under attack) and this can cause a TCP connection to be reset or an authentication error.

In most of the attacks that we analyzed, the size of the execution window between attack injection and attack detection is on the order of millions of instructions. In our VM, this lasts a few seconds, but for a CPU with a GHz clock speed this takes a small fraction of a second. It is important to point out that for the case where the system was running a Web server, our experiments were done in a local area network. In real-world attacks that might take place across continents, it could be the case that the time between attack injection and detection lasts longer in clock time because of TCP round trip time. However, for this case we could compress the idle periods in replay by identifying the idle loops and fast-forwarding the VM's CPU tick whenever we detected that the idle loop was being executed.

We have also noticed that the semi-replay phase executes less instructions than its corresponding window during normal execution (attack injection to attack detection). This is because during semi-replay we remove malicious events from the system and, consequently, all its corresponding instructions. Also, we enter normal mode as soon as we replay the last non-malicious network packet in the log fi le. In pure replay there may be other events to be processed after the last network packet is consumed. Table 2 shows, for each exploit, the number of instructions removed from the semi-replay phase.

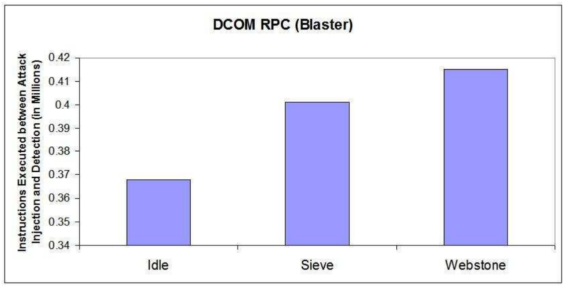

Figure 11: DCOM RPC (Blaster)

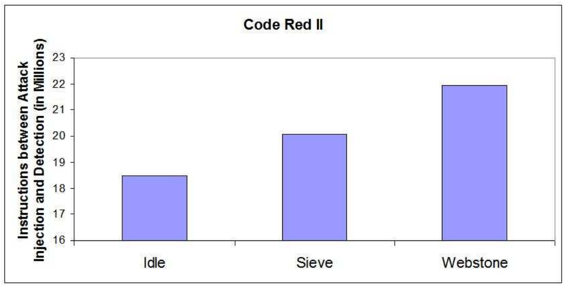

Figure 12: Code Red II

The symbolic execution data directly related to the attack and collected by WormHealer can also help the analysis of the exploit and its corresponding vulnerability. Table 3 summarizes the data collected for each exploit. From these results we can see that the expressions directly related to format string exploits have larger values for their depth and tend to be associated with a great number of Labels. Also, they are all of type Operation. The expressions directly related to the Buffer Overflow exploits have depth 1, are associated with a small number of Labels and, with exception of Code Red II, are of type Label.

Exploit	Instructions Removed from Semi-Replay
DCOM RPC (Blaster)	6627
IIS (Code Red II)	130
wu-ftpd Format String	25758
rpc.statd (Ramen)	644
innd	10145

Table 2: Number of Instructions Removed from Semi-Replay

Exploit	Number of Expressions	Type	Depth	Number of Labels	Vulnerability
DCOM RPC (Blaster)	4	Label	1	10	Buffer Overfbw
IIS (Code Red II)	2	Operation	1	12	Buffer Overfbw
wu-ftpd Format String	2	Operation	3504 and 595	12895	Format String
rpc.statd (Ramen)	4	Operation	69 (average)	492	Format String
innd	4	Label	1	10	Buffer Overfbw

Table 3: Symbolic Execution Data

3.2.2 Can Client Requests be Fulfilled?

Here we analyze how WormHealer affected the communication of client and a recovery-enabled server. We have considered the Web application where the Web server suffered a remote attack while servicing requests from remote clients. Our clients were represented by the WebStone benchmark and our Web server (in a noisy campus network) was Apache 1.3 for Linux and Apache 2.0 for Windows XP and Whistler. In order to analyze how our approach affected client/server communication, we froze the client VM while the Web server was under the recovery process. This is because our symbolic execution tool can increase the execution time of a replayed run as much as fi ve times and if we did not freeze the client VM, the application at the client side could time out. It is important to point out that our main goal here is not performance because we do not expect good performance out of a basic interpretation VM. We want to show the feasibility of an approach that can be practical if implemented in hardware or in a low-overhead VM.

Except for Windows Whistler, we have simulated 10 simultaneous clients fetching several fi les of different sizes (from 500 bytes to 5 MB). Our Web server in Windows Whistler could not handle well more than 1 client simultaneously. Each client kept making requests to the server for 1 minute. In each experiment, for each one of the exploits, as soon as the clients started making their requests, we launched the attack. After WormHealer recovered the server we unfroze the client VM and let the clients fi nish their requests. Table 4 show the results we have obtained. For Windows Whistler the results reflect only 1 Web client.

We have analyzed all TCP packets exchanged by clients and server to fi nd the cause for the errors in certain connections. We have observed that there was a connection error every time the server sent a TCP segment of type SYNACK to the client during the semi-replay phase. A SYNACK segment is part of the TCP three-way handshake and carries the initial server sequence number. During semi-replay, the number chosen by the server is different from the number it chose during normal execution because the entropy of the system changed during semi-replay. As WormHealer replays all non-malicious network packets, the

Exploit	Number of Connections Tried	Number of Errors
DCOM RPC (Blaster)	7	1
IIS (Code Red II)	6	0
wu-ftpd Format String	34.75	3
rpc.statd (Ramen)	49	0
innd	70	2.25

Table 4: Number of Client Connections x Number of Errors during Recovery

server, after sending the SYNACK will receive an ACK segment from the client where its ACK number reflects the initial sequence number the server chose during normal execution. The recovered server, on the other hand, sees this mismatched ACK number as an error and resets the connection.

3.3 Performance

The recovery time is determined by the length of the checkpoint window. WormHealer current proof-of-concept prototype requires that the checkpoint window be replayed twice to recover the server. We have made this implementation decision because the current version of our symbolic execution tool incurs too high performance overhead to be used continuously, in normal mode. A light-weight version of our symbolic execution tool that can track down only data directly used by WormHealer could be employed during normal execution, thus limiting the recovery to one replay rather than two.

As our system is an extension of a VM that employs basic interpretation [29] and our symbolic execution tool currently incurs significant performance overhead, we have left performance out of the scope of this paper. Our main focus in on availability and showing that our approach can actually recover the server execution for all selected exploits and the client server interaction for most of the TCP connections analyzed. We also believe that the proof-of-concept prototype presented here is promising and we expect that WormHealer will be practical for use in real, online systems if implemented in hardware or in a VM with better performance.

4 Related Work

4.1 Recovery from General Failures

Recovery has been addressed for a long time in areas such as databases and fault-tolerance. The approaches used for databases [16, 20] aim to protect data integrity. In the area of software fault-tolerance the goals are to be able to restart an application after a fault, or periodically rejuvenate a system in order to avoid a fault, or roll-back and/or replay an application after a fault. This can be achieved through reboot [8, 15, 30] and its variations [2, 7], software rejuvenation [18] and roll-back recovery and replay techniques [1, 5, 11, 13, 21]. These approaches cannot be directly applied for post-attack recovery because or they do not address availability, or may allow the system receive several attacks from the same source, or may loose important system state while recovering or do not include an undo or repair mechanism. Just re-starting an application (or its modules) does not improve system availability and does not prevent that the application be attacked again. Rolling-back the application or the system to a previous checkpoint (without logging) and let it continue will not allow nondeterministic events (*e.g.* network packets and user input) to be regenerated. Replaying the application from logged files without employing a repair mechanism will just cause the attack to be replayed again.

4.2 Recovery with Repair Mechanisms

Here we discuss related work involving recovery from worms, malware and general software failures that employ some form of undo or repair mechanism.

4.2.1 Recovery from Worms and Malware

INDRA [26] is an architecture for chip multiprocessors with the goal of recovering network services after an attack. The main idea is to asymmetrically configure the processor cores in a security hierarchy. High priviledged cores isolated from the network (thus immune to attacks) monitor low priviledge cores running network services. The recovery strategy is based on an efficient incremental backup strategy that undoes changes in memory caused by an attack while servicing new requests. INDRA has the limitation of not recovering the full-system state such as file system, states related to interprocess communication, messages, signals, and resources such as files opened, processes spawned, pages allocated, etc...

Taser [14] is a recovery system that selectively recovers legitimate file system data after an attack or general damage. It reverts tainted or attack-dependent operations while preserving non-malicious ones. Its focus is on file system recovery and, consequently, it does not recover corrupted memory areas and the complete system state.

Sidiroglou et al. [27] proposes a recovery approach that prevents the recurrence of a fault or attack on an application that has already exhibited it. When the application is executed in the future, the region of code where the fault or attack occurred is executed by an instruction-level emulator. This emulator then checks for recurrences of previously seen faults before each instruction is executed. Upon detecting a fault or an attack the associated function is forced to return an error and the memory modifications caused by instructions executed in the emulator are rolled-back. This approach presents some drawbacks when compared to WormHealer: it cannot recover from zero-day control-flow hijacking exploits, it performs repair by speculatively returning an error code, and it is application-specific.

DIRA [28] and failure oblivious computing [25] represent application-level compiler solutions for the problem. DIRA is a GCC compiler extension that modifies an application source code so that it can detect buffer overflows, repair any damage caused by the attack and identify the malicious packet. DIRA cannot recover damages in the file system and the authors have not discussed how DIRA works with vulnerabilities other than buffer overflows. The second approach is a C compiler that detects invalid memory accesses. An invalid write is simply discarded, thus preventing an application being terminated, and an invalid read returns a manufactured value. This is only suited for applications that apply computations with short data propagation distances because the speculative reads may produce incorrect results.

Back to the Future [17] is a recovery framework to be used for malware like Trojan horses, spyware and viruses. It monitors and logs operations of applications regarded as untrusted by users and detects an integrity violation when a trusted process is about to read untrusted data. In this case, it automatically removes malware from the system and undoes its memory modifications. The limitation is that it depends on a human to classify an application as trusted and untrusted.

4.2.2 Recovery from General Software Failures

Rx [24] is an interesting technique to recover programs from software bugs. The idea is to rollback the program and re-execute it in a modified environment when a bug is detected. It analyzes the failure based on its symptoms and accumulated experience and apply a set of environmental changes to the new execution of the application. As these changes are tentative it does not guarantee recovery and if the application is not recovered alternate solutions such as reboot are employed. In the case of worm attacks, an environmental change would be to drop some user's requests. As far as worm recovery is concerned, WormHealer is more advantageous in several respects: it knows exactly which user requests are malicious, what changes are necessary to revert the effects of the attack, guarantees recovery, does not requires changes in the application and OS, prevents attacks from the same source to occur again, and recovers the complete system state.

Brown and Patterson [6] propose the 3R's (Rewind, Repair and Replay) model to provide a mechanism that allows human operators to recover from their mistakes. In the rewind step all system state is reverted to its contents in an earlier state. In the repair step, the operator makes the necessary changes. In the replay step the system is re-executed with the changes made during the repair step. This model addresses the problem of recovery from mistakes made by humans where a human is also the entity responsible for detecting that an error has occurred and how to repair it. Our approach is different from 3R's in that our repair step is done simultaneously with the replay step and is completely automated.

BackDoor [31] is a recovery system that detects when an OS is unresponsive and salvage critical software state in the OS memory so that another recovery node, executing the same network application, can take over the client sessions serviced by the failure node. The goal is not to recover the node after failure but to be able to continue servicing its client sessions in spite of the failure.

Other related works are the runtime systems DieHard [3] that tolerates memory errors, and Exterminator [22] that detects and corrects heap-based memory errors.

5 Conclusions and Future Work

In this work we presented WormHealer, a replay-based, architecture-level post-attack recovery framework using VM technology. After a control-flow hijacking worm attack has been detected by our detection system, we replay the checkpointed run using symbolic execution to discover the source of the malicious attack. We then replay the run a second time but ignore inputs from the malicious source.

Our approach is promising because for all exploits analyzed we could discover the source of the attack. The execution and the complete system state of the WormHealer-enabled server were recovered for all cases and the communication between the server and any non-malicious client during the attack was recovered for most TCP connections. Only the connections that depended on the entropy of the system being the same as it was at the time the attack was stopped could not be recovered.

We expect that implementing the WormHealer approach in hardware or in a low-overhead VM will make it practical for use in real, online systems. As future work we plan to develop a light-weight version of our symbolic execution tool that can track down only data directly used by WormHealer. If the overhead of tracking data is suffi ciently low, we can employ it during normal execution and limit the recovery to one replay rather than two.

We also plan to implement a TCP translation scheme to avoid the entropy-related problem we experienced when the WormHealer-enabled server had to choose an initial sequence number for a TCP connection during the semi-replay phase.

References

[1] ALVISI, L. *Understanding the Message Logging Paradigm for Masking Process Crashes*. PhD thesis, Cornell University, 1996.

[2] BAKER, M., AND SULLIVAN, M. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. *USENIX* (June 1992).

[3] BERGER, E. D., AND ZORN, B. G. DieHard: Probabilistic Memory Safety for Unsafe Languages. *PLDI* (June 2006), 158–168.

[4] BIBA, K. J. Integrity Considerations for Secure Computer Systems. In *MITRE Technical Report TR-3153* (Apr 1977).

[5] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-Based Fault Tolerance. *ACM TOCS 14*, 1 (February 1996), 80–107.

[6] BROWN, A. B., AND PATTERSON, D. A. Rewind, Repair, Replay: Three R's to Dependability. *10th ACM SIGOPS European Workshop* (2002), 70–77.

[7] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot – A Technique for Cheap Recovery. *OSDI* (December 2004).

[8] CHOU, T. C. Beyond fault tolerance. *IEEE Computer 30*, 4 (1997), 31–36.

[9] CRANDALL, J. R., AND CHONG, F. T. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *MICRO* (December 2004), 221–232.

[10] CRANDALL, J. R., SU, Z., WU, S. F., AND CHONG, F. T. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. *ACM CCS* (November 2005), 235–248.

[11] CUI, W., PAXSON, V., WEAVER, N., AND KATZ, R. H. Protocol-Independent Adaptive Replay of Application Dialog. *NDSS* (February 2006).

[12] de Oliveira, D. A. S., Crandall, J. R., Wassermann, G., S. Felix Wu, Z. S., , and Chong., F. T. ExecRecorder: VM-Based Full-System Replay for Attack Analysis and System Recovery. *Workshop on Architectural and System Support for Improving Software Dependability (ASID)* (October 2006), 381–390.

[13] Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *University of Michigan Technical Report CSE-TR-410 34*, 3 (September 2002), 375–408.

[14] Goel, A., Po, K., Farhadi, K., Li, Z., and de Lara, E. The taser intrusion recovery system. *ACM SOSP* (2005), 163–176.

[15] Gray, J. Why do computers stop and what can be done about it? *5th Symp. on Reliability in Distributed Software and Database Systems* (1986).

[16] Gray, J., Helland, P., O'Neil, P., and Shasha, D. The Dangers of Replication and a Solution. *ACM SIGMOD* (1996), 173–182.

[17] Hsu, F., Chen, H., Ristenpart, T., Li, J., and Su, Z. Back to the Future: A Framework for Automatic Malware Removal and System. *ACSAC* (December 2006), 163–176.

[18] Huang, Y., Kintala, C., Kolettis, N., , and Fulton, N. D. Software Rejuvenation: Analysis, Module and Applications. *FTCS-25* (June 1995), 381–390.

[19] King, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (1976), 385–394.

[20] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: a transaction recovery method supporting fi ne-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS 17*, 1 (March 1992).

[21] Newsome, J., Brumley, D., Franklin, J., and Song, D. Replayer: Automatic Protocol Replay by Binary Analysis. *ACM CCS* (November 2006).

[22] Novark, G., Berger, E. D., and Zorn, B. G. Exterminator: Automatically Correcting Memory Errors with High Probability. *PLDI* (June 2007), 158–168.

[23] Patterson, D. A., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., and Treuhaft, N. Recovery oriented computing (roc): Motivation, defi nition, techniques, and case studies. Tech. rep., UC Berkeley - Technical Report UCB//CSD-021175, 2002.

[24] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. *ACM SOSP* (October 2005), 235–248.

[25] Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., and William S. Beebee, J. Enhancing Server Availability and Security through Failure-Oblivious Computing. *OSDI* (2004).

[26] Shi, W., Lee, H.-H. S., Falk, L., and Ghosh, M. An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processor. *ISCA-33 34*, 2 (May 2006), 102–113.

[27] Sidiroglou, S., Locasto, M. E., Boyd, S. W., and Keromytis, A. D. Building a Reactive Immune System for Software Services. *USENIX - Annual Technical Conference* (April 2005).

[28] Smirnov, A., and cker Chiueh, T. DIRA: Automatic Detection, Identifi cation, and Repair of Control-Hijacking Attacks. *NDSS* (February 2005).

[29] Smith, J. E., and Nair, R. *Virtual Machines - Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[30] Sullivan, M., and Chillarege, R. Software defects and their impact on system availability - a study of fi eld failures in operating systems. *21st International Symposium on Fault-Tolerant Computing* (1991).

[31] Sultan, F., Bohra, A., Gallard, P., Neamtiu, I., Smaldone, S., Pan, Y., and Iftode, L. Recovering Internet Service Sessions from Operating System Failures. *IEEE Internet Computing 9*, 2 (March/April 2005), 17–27.

[32] UnixBench. `http://www.tux.org/pub/tux/benchmarks/System/unixbench/`.

[33] Webstone 2.5. `http://www.mindcraft.com/webstone/`.

[34] bochs: the Open Source IA-32 Emulation Project (Home Page). `http://bochs.sourceforge.net`.

[35] Security Focus Vulnerability Notes, (http://www.securityfocus.com), bid == Bugtraq ID.