

# Effective Recovery from Security Violations Using Reversible Computing

Chris Hazard, Seongbeom Kim, and Brian Rogers  
May 9, 2005

## 1. Introduction

Attacks on computer systems come in many forms and have ranging levels of severity. As a result many different types of security mechanisms exist, each of which attempts to prevent or detect some set of these attacks. Some of these security mechanisms are dynamic, meaning that they try to detect and stop attacks as they happen during the execution of an application. For example, a dynamic security mechanism to prevent buffer overflow attacks may monitor all string buffer operations and dynamically check the buffer bounds before each operation to make sure that it is only operating within the bounds of the string. Typically, if this type of security mechanism detects that an operation will overflow a buffer; it will simply terminate execution of the application.

While these types of dynamic security mechanisms can protect the computer system by preventing attacks such as buffer overflows, they create a new problem that results as a side effect of terminating the application. Though they prevent the system from being controlled by the attacker, terminating the application is still a denial of service (DoS) attack. Even though servers usually restart processes, pending data may be lost. Repeated attacks can make other users' performance severely degraded at best. The user will not be able to make forward progress because the application will constantly have to restart. This also wastes CPU resources that could be used for running other services, as service initialization is often more costly than sustained execution.

Many dynamic security mechanisms have strong security properties, and can stop many types of attacks, but it would be desirable to have a more graceful way to recover from these attacks. Buffer overflows and some other attacks can be caused by an attacker supplying some input to a program that is malicious or "tainted". By the time an operation is reached that would cause a buffer overflow with the input, it may not be known where the tainted input came from, or if any other safe value could be used in place of the tainted one. This leaves no option but to simply exit the program and begin its execution again. If a method existed to obtain a new valid value and undo any harm done by this operation, or to throw away the malicious input altogether, then it may be possible to safely complete the operation and finish executing the application without aborting. In this paper we study the feasibility and implement a

system that is capable of doing this. With the aid of a reversible computing system, we can “rewind” the operations of an application from the point where a dynamic detection scheme detects an attack to the point where the tainted input that resulted in the attack was obtained. Then we can throw this input away and continue execution back in the forward direction with a new input.

The remainder of this paper is organized as follows. Section 2 presents related work on different types of system recovery that pertain to our approach. Section 3 discusses the details of how our approach works and how it was implemented. Section 4 evaluates the implementation of our approach on some test cases. In section 5 we discuss some of the important issues that need to be addressed with regard to our proposed system, and we give our conclusions in section 6.

## **2. Related Work**

### **2.1 Reversible Computing**

Reversible computing is the ability of a computing system to revert back to any previous system state from any system state. This does not necessarily mean that all system information is stored for every instruction, but only the information required to make each instruction revert the system back to the state before it executed. Many instructions, such as adding a constant, are inherently reversible, requiring no such extra information.

Intuitively, reversible computing lends itself well for reversible debugging [3, 4, 5]. This feature alone would not merit a commoditization of natively reversible computers. However, the quantum and Newtonian physics of our universe underlying all of computing is itself reversible. It has been long known between the fields of information theory and computer architecture that a large portion of dynamic power consumption in processors is caused by the destruction of information, fighting the reversibility of physics [1]. Regardless of the quality of transistors, this power consumption is inevitable in non-reversible computing. Using adiabatic physical processes, reversible computing could keep the power consumption and heat dissipation low while delivering high computational throughput. Additionally, many types of quantum computers will require a reversible design in order to operate [6].

Because of the potential for quantum computing, power consumption benefits, and debugging value, reversible computing has been receiving more attention lately. Much of this recent work started with a group at MIT in the mid 1990’s. Vieri demonstrated a practical reversible computer architecture called “Pendulum” in his 1995 MS thesis [2], and Frank investigated the implications of programming a reversible computer in his 1999 PhD thesis [1]. All irreversible computations may be emulated by a reversible computer, and the upper bounds on the computational overhead in terms of both memory and

algorithm complexity have been found [7]. Specialized Instruction Set Architectures have also been explored for minimizing the amount of energy dissipation [15].

It may not be necessary or practical to have a fully reversible computer [1]. In lieu of full reversibility, systems may perform checkpointing operations [8], or simply destroy old reverse information in a manner that would minimize heating of the processor and attempt to reuse energy. Reversible computing has been investigated for use in determining fail-safety in software design [16], but to our knowledge has never before been used as a recovery method for system-level reliability and security.

In our proposed system, untrusted input will have special status. When reversing from a fault back to the source of last input, the questionable input will be destroyed. As discussed in [1], it is possible to build an operating system capable of reversing an individual process. In this case, only the malicious data sent to the affected process would be discarded.

## **2.2 Checkpointing**

Checkpointing is the act of saving the state of a running program so that it may be reconstructed later in time. There are three levels where checkpointing can be implemented. They differ in the level of user/programmer involvement.

1. OS checkpointing: Here, checkpointing is performed by the operating system. Typically, any program can be checkpointed by the operating system without any effort on the part of the programmer or user. Standard process preemption can be viewed as a simple form of OS checkpointing [9][10].

2. User-level, transparent checkpointing: Here, checkpointing is performed by the program itself. Transparency is usually achieved by compiling the application program with a special checkpointing library. Since checkpointing is performed on top of, rather in the operating system, the recoverability of operating system state is an important issue. Programs must assume that their process ids may change over time as the result of being checkpointed and restored [11][12].

3. User-level, non-transparent checkpointing: Here, programmers actively incorporate checkpointing into their programs, often with the help of libraries and preprocessors. Non-transparent checkpointing obviously places a much larger burden on the programmer. The tradeoff is in performance and flexibility. Programmers can specify the exact information that is needed for recovery, and thus checkpoint less information than transparent checkpointers.

There are several uses of checkpointing. The major use for checkpointing is fault-tolerance. [13][14] This is typically called checkpointing with rollback recovery. At a periodic interval, the

application stores checkpoints to disk. If a failure occurs that causes the application to be terminated prematurely, the application can restart from its most recent checkpoint, losing at most an interval's worth of computation.

Process migration is another use of checkpointing. Instead of storing a checkpoint to disk, the checkpointing processor sends its checkpoint to another processor, which begins its computation from this checkpointed state.

Checkpoints may also be stored for purposes of debugging a program. For example, most debuggers have tools for examining checkpoints, which are created when a program exits abnormally.

When a program is executing, its state is composed of the values in memory, the CPU registers, and the state of the operating system (including the file system). Usually, the memory is divided into four parts: code (or text), the global variables (or data), heap and stack. Of these, the global variables, heap and stack need to be stored along with the registers in a checkpoint. Typically, the code is unchanged from the program's executable file, and thus may be restored from there in the event of a failure.

If the checkpointer is implemented in the operating system, then enough information can be stored with the checkpoint to restore the view that the program had to the operating system at the time of the checkpoint. If the checkpointer is implemented at user level, then it does not have the privileges to restore the operating system, but instead can attempt to make the operating system appear as it was at the time of the checkpoint.

A transparent checkpointer should be able to rebuild as much state that is external to the checkpointing process as possible. Besides operating systems internals and the file system, other external states that can be reconstructed include the window system and the state of external servers.

Most non-transparent checkpointers give the programmer primitives for storing and recovering data that is in the global variables, stack and heap. However, the programmer is responsible for restoring the execution state of the program upon recovery. Although this places a greater burden on the programmer, it can afford functionalities such as restoring the checkpoint on a machine of differing architecture from the checkpointing machine. This is because machine-dependent details such as memory layout and the definition of stack frames are not part of the checkpoint.

It seems reasonable to use checkpointing for recovering from security attacks. While existing methods usually crash the program, it can graciously roll back to the last checkpointed state. However, checkpointing machine and program state may take significant overhead and require large reliable storage. Also, periodic checkpointing may not be fine-grained enough if we want to do rollback to exact place where the error has occurred.

### **3. Our Approach**

#### **3.1 Main Idea**

The central idea of our approach is to use reversible computing with a dynamic security detection mechanism (e.g. Stackguard [17] for detecting buffer overflows) so that we can recover more efficiently from detected attacks and prevent DoS situations from resulting. Our reversible system operates in the following way. Execution proceeds as normal in the forward direction at the start of an application, and it is monitored by the dynamic detection mechanism. If a security fault is detected, such as a buffer overflow that we consider in our implementation, the reverse mode of execution is initiated. Through the use of reversible computing, we can execute each instruction of the program in reverse until we reach the point where the previous external input was obtained. Then we discard this input, and obtain a new one. Finally we continue execution in the forward direction again and if this new input is a valid one then we will make further progress in the application without having to crash it.

Even though the system is able to purge itself of malicious inputs, DoS attacks are not impossible, just much more difficult. It takes a finite amount of time for the system to reverse execute the program back to the point where the malicious input was received in order to purge itself of the message. If the malicious messages are sent at a rate faster than the system is able to purge them, with much more malicious input than legitimate input, then performance may be degraded enough to be considered a DoS. However, the rate of malicious messages required to cause this effect would most likely be very high, in the same manner as traditional traffic flooding DoS attacks.

#### **3.2 Implementation**

We have implemented three main items in order to study this approach. Essentially with our implementation we wanted to determine the feasibility of using reversible computing for our purposes, and we wanted to understand some of the issues that must be addressed in order to implement a real-world system of this type. First, we needed to choose an Instruction Set Architecture (ISA) that was very functional, but that we could reverse instruction by instruction. The main tradeoff is that traditional ISAs require larger history buffers whereas reversible ISAs require more work on writing the assembly code (very few compilers are capable of generating code for a reversible ISA). We found that the standard MIPS ISA works well for our purposes. In our implementation, we included all MIPS instructions with the exception of floating point instructions and a few less recently used instructions.

Secondly, we needed a way to execute these instructions in the forward and reverse direction to model our system. We implemented an interpreter in Perl that could execute all of our ISA along with some system calls to obtain external input and print output. The interpreter has the ability to change from



```

30     .globl proc_msg
31 proc_msg:
32     subi $sp, $sp, 4
33     sw   $ra, 0($sp)           # push $ra to the top of stack
34     addi $s1, $0, 0xcafebabe
35     subi $sp, $sp, 4
36     sw   $s1, 0($sp)         # push canary value to the stack
37     subi $sp, $sp, 16        # allocate 16 byte buffer
    ...
    # call get_msg to get input
    ...
    # call prn_msg to print output
    ...
60     addi $sp, $sp, 16        # adjust stack before return
61     lw   $s1, 0($sp)
62     addi $sp, $sp, 4         # pop canary from the top of stack
63     addi $s2, $0, 0xcafebabe
64     beq  $s1, $s2, Pass     # check canary value
    ...
68     li   $v0, 10           # terminate program
69     syscall
70 Pass:
71     lw   $ra, 0($sp)
72     addi $sp, $sp, 4         # pop $ra from the TOS
73     jr   $ra               # return

```

The micro-application in MIPS assembly

Once ‘main’ calls ‘proc\_msg’, the return address is pushed onto the stack and then pushed the pre-determined canary constant, 0xCafEBaBe, is pushed on top of it. To allocate the 16-byte local buffer, the stack pointer is decremented accordingly (line 32~37). At this time, the stack will look like the following.

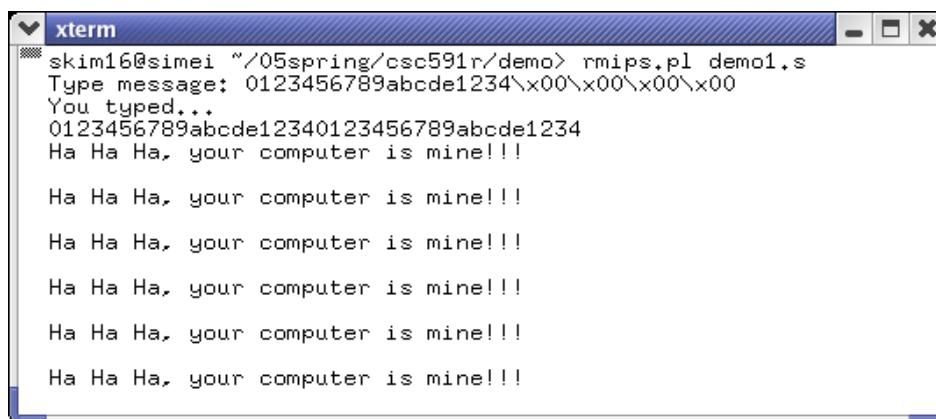
buf[0...3]
buf[4...7]
buf[8...11]
buf[12...15]
canary value (0xcafebabe)
return address

When attackers overflow the buffer by providing a longer message than 16 bytes, they may tamper with the return address but cannot avoid mangling the canary value. By checking the integrity of the canary value, we can dynamically detect whether there is a stack overflow attack. This is the main idea of StackGuard [17] which we adopted in this evaluation. Line 62 through 64 pops the canary from the stack and checks the sanity. If the canary value is intact, the return address from the stack is used to

return. Otherwise, existing scheme would terminate the program (line 68~73).

Instead of terminating the program, our scheme can trigger reverse execution until we reach the malicious input. To trigger the reverse execution, we defined a new instruction, 'reverse'. Line 68, 69 of the above example is replaced by 'reverse' to support our reverse-execution recovery mechanism.

We presented three screen-shots below, each of which shows the result from a different scenario. The first screen-shot shows the result when the stack overflow attack succeeds. We can see that the control flow is tampered by malicious input. The second screen-shot shows that the dynamic detection scheme prevents the attack by terminating the program. However, we need to restart the program to serve further requests from valid users. Finally, the third screen-shot shows the result from our recovery scheme. Instead of terminating the program, the program is executed backward until new input can be typed again.



```
xterm
skim16@simei ~/05spring/csc591r/demo> rmips.pl demo1.s
Type message: 0123456789abcde1234\x00\x00\x00\x00
You typed...
0123456789abcde12340123456789abcde1234
Ha Ha Ha, your computer is mine!!!

Ha Ha Ha, your computer is mine!!!

Ha Ha Ha, your computer is mine!!!

Ha Ha Ha, your computer is mine!!!

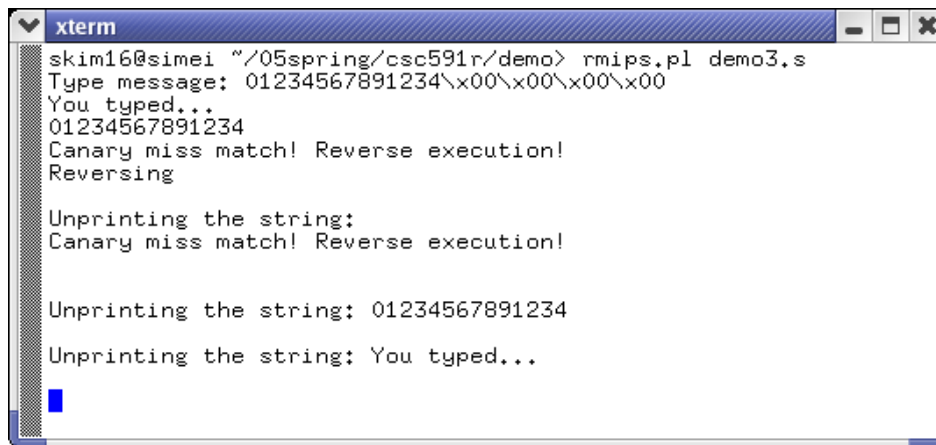
Ha Ha Ha, your computer is mine!!!

Ha Ha Ha, your computer is mine!!!
```



```
xterm
skim16@simei ~/05spring/csc591r/demo> rmips.pl demo2.s
Type message: 01234567891234\x00\x00\x00\x00
You typed...
01234567891234
Canary miss match! Terminate program!
skim16@simei ~/05spring/csc591r/demo>
```





```
xterm
skim16@simei ~/05spring/csc591r/demo> rmips.pl demo3.s
Type message: 01234567891234\x00\x00\x00\x00
You typed...
01234567891234
Canary miss match! Reverse execution!
Reversing

Unprinting the string:
Canary miss match! Reverse execution!

Unprinting the string: 01234567891234

Unprinting the string: You typed...
```

## 5. Discussion

### 5.1 Improving Recoverability Coverage

Even though our method can prevent immediate attacks from occurring, it is still quite possible that the program will become stuck in a loop, causing a denial of service. Consider the following example function behavior:

1. Read string into buffer1
2. Read string into buffer2
3. Perform some operations
4. Verify stack integrity (check canary value)
5. Return from function

In this example, if the buffer overflow was exploited with buffer2, our security method will recover properly. The integrity check will fail at step 4, the system will reverse back to step 2, and await buffer2 to be read in again. However, if buffer1 was exploited instead of buffer2, then the system has no way of knowing to revert all the way back to step 1. The system will revert back to step 2, re-read in buffer2, and fall victim to the same exploit regardless of what is read in for buffer2. Because this causes the application to fall into an infinite loop, it is effectively worse than a denial of service which simply terminates the program. In falling into an infinite loop, the program is using CPU resources that could be used by other applications, and additionally could not be automatically respawned without a sophisticated process monitor. We will refer to this specific vulnerability as a *latent input vulnerability*.

We present both a sophisticated and simple way of addressing this issue, which can be used

together for maximal security protection. The sophisticated method increases the coverage of vulnerabilities that can be recovered by reversible computing, while the simple method reduces the harm done by vulnerabilities that penetrate the system's recovery capabilities.

The sophisticated method borrows conceptually from taint checking. Taint checking is the method of marking input as not trusted (tainted), and propagating the taint flag to any other data that is affected by it. For example, if a tainted value is added to an untainted value, the result is tainted. The system can then check if a value is tainted before using it in vulnerable situations, such as setting the program counter to a dynamically computed address. In order to know how far back for the processor to reverse, every datum (may be byte, word, cache line –many sizes have proven successful in taint checking) needs to maintain which inputs have affected it. When two tagged data are combined, the result is tagged with the combination of the input tags of both data. When an attack is detected, the processor can reverse back to the earliest input that possibly influenced the exploit, purging itself of all the subsequent input (see section 5.3 for a discussion as to how this excess purging can be made less detrimental). We will refer to these input markers as *tags*, and refer to data affected by input as *tagged data*.

With or without reversible computing, it is quite infeasible to maintain every input that has affected a datum, so this must be constrained. It is only feasible to retain a small number of the most recent inputs for tag propagation. Retaining the 4 most recent inputs would allow for the successful purging of an attack that had a latent input vulnerability with 3 or fewer inputs between the malicious input being received and the exploit detected. By retaining a small number of the most recently inputs for tagging, such as 3 or 4, all but the most difficult and complex exploits against latent input vulnerabilities can be recovered. To keep a truncated history, old input tags must be discarded. This can be done by indexing inputs by tag offset, and treating the tag bits as a shift register. For example, tagging the previous 4 inputs would require 4 additional bits per datum, and the most recent input tag would be the first of the 4 bits. When a new input is received, all tags throughout the memory hierarchy would be shifted by one bit. This approach allows simple hardware combining of tags in an operation, as only a bitwise OR is required. The main drawback of this approach is that a signal line to all of the memory hierarchy is required in the architecture. We have not been yet come up with a more effective solution that does not have such a signal line requirement in some capacity.

Aggressive tagging in the architecture would provide a conservative approach to purging input. For example, the program counter (PC) should be tagged when an instruction branches based on the comparison of a tagged value, and the PC's tags may potentially be used to tag all instructions. If an

exploit is detected with untagged data, then it would be safe to assume that the malicious input came earlier than the beginning of the tagged input history. In this case, the program should abort, as it cannot reverse far enough.

The second and simple method to prevent latent input vulnerabilities from being exploited is to keep a counter for each of the currently tagged inputs (only the most recent input if tagging is not used). Every time a new input is received from a reverted system read, the counter for that read is incremented. If the counter reaches a certain specified threshold, then the program will be aborted. This way, if an attack is too complex for our security method to prevent, the attack will do not cause any more harm than if reversibility were not utilized. Using a threshold only decreases the effectiveness of the reversibility slightly; if more malicious messages are being received than legitimate traffic, then the application may still be frequently aborted. Increasing the threshold value can be used to counter this and prevent the process from being aborted. Interestingly, if a particular high-volume attack on a service-based program is determined to not be a latent input vulnerability and the administrator does not wish to raise the threshold value for fear of other latent input vulnerabilities, the process can still be prevented from aborting. Inserting inert messages, possibly even erroneous, into the program's input would prevent the counters from continually increasing from malicious messages, thus preventing the program from aborting.

## **5.2 OS Implications**

Instead of assuming a purely reversible computer, our approach assumes that the reversible computer has the ability to reverse individual processes; different processes running simultaneously may be running in different directions. This is to allow the operating system to behave differently when executing in reverse than it would when executing forward, which is required in order to provide the functionality to purge malicious inputs. We also assume the paradigm discussed by Michael Frank involving processes recalling output sent to other processes [1]. When a process reverses an output, it reverses the execution direction of the process it sent the message to until the message is purged from that process. This will be discussed further in section 5.3.

Given that the operating system kernel is always running in the forward direction, it needs to have both forward and reverse implementations of all of its services. Both software and hardware interrupts change the processor to run forward execution on the corresponding kernel function. To support the reversal of a kernel function executing in the forward direction, a combination of the instruction set architecture and kernel code must obtain the parameters, return values, and state

information from the kernel call when it happened in the forward direction.

Because each process can be running in either the forward or reverse direction, each process must have its own execution history buffer. Realistically, the history buffers will be finite in length, so the process can only reverse execute back to the beginning of the buffer. The oldest history data would be continuously destroyed to make room for the new history data. If the process attempts to reverse back to a point prior to the start of the history buffer, then the process must be aborted.

### **5.3 Implied Message Paradigm**

We initially considered implementing both a message-based and stream-based reversible input system call. The message-based implementation would read input line-by-line, while the stream-based implementation would read input character-by-character. We soon realized that the major framing (message misalignment) implications that arise when part of a stream is lost would make purging malicious input very difficult. Designing a system based around messages and events rather than streams is not a too constraining, as Microsoft Windows uses this approach (as opposed to Unix architectures, which are more stream-based). One major implication of using the reversible architecture is that all applications using these message communications need to be able to handle the “take-back” of a message. Most systems will support this transparently, as the operating system can throw the targeted process in reverse to purge it of the message taken back. However, all network protocols and peripherals must also support the ability to “take-back” messages.

In systems of higher interdependence, this cascade of message “take-back” would throw much of the system into reverse execution until the earliest relevant input was purged, then resume and reconstruct execution. From a user-interface perspective, it may be useful to have a separate part of the interface, perhaps closely tied to the kernel, dedicated to remembering input that has been purged. As long as it keeps the transaction information on a per-program basis, this interface could resend messages purged from other applications or machines on the network. By allowing the user to manage communications to the machine, the user could be immediately notified of any information rejected by almost any type of error. This would help prevent user frustration caused by information purged due to errors by allowing the user to automatically reenter the data.

## **6. Conclusion**

We have shown that future reversible computing architectures could allow us to more effectively recover from dynamically detected attacks by preventing DoS scenarios. In addition, we have identified and addressed some key issues that must be addressed in a real-world system of this type. We presented

two alternative schemes to more robustly handle attacks that result from different input sources through a type of taint analysis and/or counters to limit the maximum number of reversals per input. We also analyzed some of the main OS issues and the extra functionality that must be provided for a reversible system that is capable of handling multiple processes simultaneously. Finally, we made a case for a message-based paradigm that would make things more feasible for reversible systems in handling malicious inputs. It appears that reversible computing offers promising ways to recover from various types of security attacks by allowing us to recover efficiently and prevent attacks from turning into DoS situations.

## REFERENCES

- [1] Michael P. Frank. Reversibility for Efficient Computing. PhD thesis, Massachusetts Institute of Technology, 1999.
- [2] Carlin J. Vieri. Pendulum: A reversible computer architecture. Master's thesis, MIT Artificial Intelligence Laboratory, 1995.
- [3] S. Lewis. Techniques for Efficiently Recording State Changes of a Computer Environment to Support Reversible Debugging. Master's thesis, University of Florida, 2001.
- [4] T. Akgul and V. J. Mooney. Instruction-level reverse execution for debugging. Technical Report GIT-CC-02-49, Georgia Institute of Technology, September 2002.
- [5] B. Biswas, R. Mall. Reverse execution of programs. ACM SIGPLAN Notices, 34(4). pp 61-69. 1999.
- [6] T. Hey. Quantum computing: an introduction. Computing & Control Engineering Journal, 10(3). pp 105-115. 1999.
- [7] H. Buhrman, J. Tromp, P. Vitanyi. Time and Space Bounds for Reversible Simulation. Proc. International Conference on Automata, Languages and Programming, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2001
- [8] A. Griewank, A. Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. ACM Transactions on Mathematical Software (TOMS), 26(1). pp 19-45. 2000.
- [9] B. A. Kingsbury and J. T. Kline, Job and process recovery in a UNIX-based operating system. In Conference Proceedings, Usenix Winter 1989 Technical Conference, pages 355-364, San Diego, CA, January 1989.
- [10] M. Russinovich and Z. Segall. Fault-tolerance for off-the-shelf applications and hardware. In 25th International Symposium on Fault-Tolerant Computing, pages 67-71, Pasadena, CA, June 1995.
- [11] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, pages 313-325, Orlando, FL, June 1994.
- [12] J. Long W. K. Fuchs and J. A. Abraham. Implementing forward recovery using checkpointing in distributed systems. In 2nd IFIP Working Conference on Dependable Computations for Critical Applications, pages 20-27, February 1991.
- [13] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In Conference Proceedings, Usenix Winter 1995 Technical Conference, pages 213-223, January 1995.

- [14] Y. Huang, C. Kintala, and Y-M. Wang. Software tools and libraries for fault tolerance. IEEE Technical Committee on Operating Systems and Application Environments, 7(4):5-9, Winter 1995.
- [15] J. S. Hall. A reversible instruction set architecture and algorithms. Physics and Computation, pp 128--134, 1994.
- [16] P. Bishop. Using Reversible Computing to Achieve Fail-safety. ISSRE 97, IEEE Computer Society Press, 1997.
- [17] C. Cowan, C. Pu, and et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. Proc. 7th USENIX Security Conf, 1998.