

# Compiler Support for Automatic Undo Code Generation

Alexey Smirnov    Tzi-cker Chiueh

Computer Science Department

State University of New York at Stony Brook

Stony Brook, NY 11794-4400, USA

Phone: 1-631-632-8436    Fax: 1-631-632-8334

E-mail: {alexey, chiueh}@cs.sunysb.edu

## Abstract

To repair a computer system or an application from a human error, a software/hardware failure or a malicious attack typically requires the ability to undo the associated damaging side effects. Although the principles for implementing this repair logic are well known, developers have to manually codify them into their programs and spend extra time and efforts testing and debugging the resulting undo code. This paper describes the design, implementation, and evaluation of a novel compiler that can automatically generate the undo code given high-level user annotations, thus significantly reducing the development efforts required for fault- and intrusion-tolerant applications. This Automatic Program Repair (APR) compiler is implemented as an extension to GCC 3.4.1, and is able to instrument the source code of an arbitrary C program in such a way that the program can restore its memory image and file system state to any arbitrary point during the course of its execution. Essentially, the APR compiler frees the programmer to focus only on the "do" logic and automatically takes care of the "undo" part. To demonstrate the effectiveness of the APR compiler, we apply it to two applications areas. First, we use the APR compiler to protect applications that are being compromised by buffer overflow attacks. In addition to repairing themselves, programs compiled by the APR compiler can detect a buffer overflow attack and identify the packets responsible for the attack. Second, we use the APR compiler to automatically add an undo capability to interactive applications such as a text editor. Using such an undo command, the user can roll back the erroneous actions that she performed. As for the efficiency of the undo code that the APR compiler generates, measurements show that it adds a 25% run-time overhead to network daemons and a negligible overhead to interactive applications.

**Keywords:** Intrusion Detection/Tolerance, Software and System Reliability, Security, Case studies and issues in the design of computer and communication systems, Code transformation.

**Submission category:** Regular Paper.

**Approximate word count:** 9,400

*The material has been cleared through our institution.*

# 1 Introduction

The operation of a computer application can be disrupted due to various types of failures, such as a hardware/software fault, a human error, or a malicious attack. When a failure occurs, it is important to quickly repair the damaged computer application such that the application becomes available to the users. To repair a computer application, one can either roll the application's state back to a checkpoint before the failure occurs and then move it forward via a *redo* log to the state immediately before the failure, or to roll the application's current state backward step by step via an *undo* log until it reaches the state immediately before the failure. In the most general case, an application's state includes its address space, the file system state, and the cumulative history of all messages it exchanges with other processes or the operating system. In this paper, we only focus on the memory state and file system state components. We used the undo logging approach because it preserves the memory state of the compromised application and meshes very well with other techniques (such as attack detection).

When an application fails but not crashes, the side effects of the failure can be erased through an undo mechanism. A classic example of this is the case when a user performs an incorrect edit operation and cleans out its effect by undoing the operation, which in turn may require undoing other intermediate operations. As another example, when an application is compromised by an attack packet, it is desirable to undo the side effects of the attack and continue the application as if the attack never happened.

Traditionally, application developers need to build this undo mechanism into their applications to mitigate the effects of failures. That is, this post-failure repair logic is considered a necessary evil that imposes additional design/implementation efforts upon the application developers. This paper describes the design, implementation, and evaluation of a novel compiler, called *APR* (*Automatic Program Repair*), that can largely automate the process of generating a particular type of repair logic, and thus greatly reduce the manual development efforts required for building fast failure-repair applications.

The *APR* compiler can automatically generate the necessary undo code given programmer-provided annotations, and is implemented as an extension to GCC 3.4.1. The undo code added to a program is responsible for logging of all the updates to the address space and the file system state while it runs and for undoing these updates when a failure is detected. As a result, programmers only need to concentrate on the “do” part of the application logic, and leave the “undo” part to the *APR* compiler. To demonstrate the effectiveness of *APR*, we apply it to two applications, an interactive text editor and an application-specific automatic attack detection and repair system.

Before a failure can be repaired, it needs to be detected first. For certain types of failures, the failure detection logic can also be generated automatically. For example, we show in this paper that a unified memory updates logging framework can be used to detect a buffer overflow attack, identify the attack packets that caused it, and repair the attacked application. Unfortunately, automatic failure detection is not always possible. For example, detecting such human mistakes as input errors in interactive applications cannot be easily automated.

The rest of this paper is organized as follows. Section 2 reviews previous research on program repair methods and fault-tolerant systems. Section 3 outlines the repair-by-undo framework and describes the memory updates logging framework, which forms the core of undo-based logging and repair. In Sections 4 and 5 we discuss the implementation details of this

framework for two types of applications — interactive editors and network daemons respectively. Section 6 presents the performance measurements of a fully operational *APR* prototype and their analysis. Section 7 concludes this paper with a summary of major research contributions and a brief outline of the on-going work.

## 2 Related Work

Our work is based upon a broad area of survivable, fault-tolerant, and intrusion-resilient systems. We will review the relevant work on both roll-forward and roll-back based recovery approaches below.

A typical roll-forward repair strategy is based on creating periodic file state backups and then restoring them at the time of repair or duplicating data on multiple disks. RAID [11] is the first technology in which multiple inexpensive disks were used to achieve higher performance and availability.

Traditional database recovery methods have been discussed in many database textbooks [3, 15]. The basic idea behind such methods is to use write-ahead logging (WAL). The central concept of WAL is that every change to the database should first be written into a log file residing on a permanent storage before being written into the disk file containing the database. This technique eliminates the need to write the updated part of the database to the disk at the time of transaction commit. Combined with data replication, WAL presents an efficient way for a database to recover from media failures.

Berkeley Recovery Oriented Computing project [19, 5, 6] aims at designing and building highly-dependable Internet services using the 3R (rewind, repair, and replay) framework [7]. According to this model, all the external inputs that come to the system need to be logged while it is operating in the normal mode. When a software configuration or a human error is detected, the system is shut down and repaired by the system administrator. After that, the underlying file system state is rolled back to a pre-fault state. Finally, all the recorded activity is replayed assuming that the application will behave correctly this time. The undoable e-mail store [8] is a project whose goal is to build an undoable e-mail store using the 3R framework.

An alternative way of bringing a compromised system back to the normal state is a complete restart. Candera et. al. [10, 9] develop the concept of micro-reboots. According to this concept, a complex system comprised of many individual components (such as a large Internet service) can be efficiently repaired from a fault or an attack by performing a micro reboot of a single failed component rather than that of the whole system. If the problem cannot be fixed by micro-rebooting then it is deferred to human operators.

There are several examples of general-purpose solutions to the problem of program replay such as Igor [13], RECAP [18], and Flashback [21]. The underlying idea of these systems is to use the operating system techniques to add the undo capability to an arbitrary program. For example, Igor [13] is a system that saves modified memory pages at each checkpoint. RECAP [18] and Flashback [21] use copy-on-write `fork()` system call to checkpoint their execution state.

The roll-back repair approach requires logging of every operation performed by the system so that any of them can be rolled back if necessary. Wylie et al. [23, 22] describes a survivable storage system S4, which is a network-attached object store with an access interface based on storage of objects. The S4 object store maintains previous versions of storage objects in a way transparent to both operating system and applications. In other words, S4 intercepts and logs at the interface

between a file system and a network storage device. The main rationale behind this design decision is that S4's designers assumed that even the operating system itself could also be compromised.

On the commercial side, the NTFS file system used in Windows XP allows several rollback options including "last-known good configuration", device driver rollback, and complete system rollback.

The roll-back approach can also be used to protect existing off-the-shelf applications from a compromise or a human error. The RFS [24] and RDB [20] projects add a selective undo capability to an existing off-the-shelf file server and database server respectively. In more detail, the RFS project aims at improving the speed and precision of post-intrusion damage repair for NFS servers. RFS maintains file system operations log and carries out dependency analysis to provide fast and accurate repair of damage caused by NFS operations issued by attackers. Based on similar ideas, RDB aims at protecting an off-the-shelf database server from malicious transactions by recording the inter-transaction dependency information at run-time and using it at repair time. At repair time, the effect of the malicious transactions is wiped out by generating appropriate compensating transactions.

The problem of database post-intrusion recovery has also been addressed by Ammann et al. [2, 16, 17]. In particular, the authors have addressed the problem of on-the-fly database repair. In [2], Ammann et. al. propose various algorithms for recovery from malicious transactions that have different tradeoffs between repair accuracy and database availability. Based on this work, an intrusion tolerant database system [17] was implemented as an enhancement to Oracle database server.

Spyder [1] is a general-purpose program roll-back and replay system. It is based on the notion of execution history. During its normal execution, Spyder records the program counter and the old values of all variables that the current instruction will change. At repair time, these values can be used to restore the memory image and the processor state.

### 3 Repair by Undo

In this section we present the undo-based repair logic from an application-independent point of view. We also discuss the memory updates logging mechanism, which is the core of the undo-based repair framework.

#### 3.1 Overview

At compile time, the *APR* compiler instruments the source code of an input program by first inserting proper memory and file state updates logging code to keep track of the memory and file system updates, then adding failure detection code if necessary, and finally augmenting the original program with a number of special functions that repair the program when a failure is detected (manually or automatically). At run time, the instrumented program generates a memory updates log that can be used to detect a failure and/or repair itself once such a failure is detected.

The repair procedure in general proceeds in the following steps. First, the application's control is transferred to the repair procedure, which first determines the scope of repair, i.e., the amount of modifications to the memory and file system states that need to be rolled back. Then, the updates to the two states are undone. Finally, the control is transferred back to the application so that it can continue execution normally. This procedure raises the following issues:

- How are updates logged,
- How to detect a failure,
- How to determine which updates should be undone, and
- Where should the program resume its execution after repair.

Answers to these questions are different for different types of applications. As a result, some issues are addressed by the *APR* compiler and others are left to the programmers. We will discuss these issues in more detail in Sections 4 and 5. However, there is one common mechanism that is the core of all undo-based repair systems, *memory updates logging*, which is described in more detail next.

### 3.2 Memory Updates Logging

Memory updates logging makes it possible to restore the state of a failed program back to the state before the failure occurred. *APR* uses a fine-grained asynchronous checkpointing approach to log updates to global and static variables to a reserved part of memory called *memory updates log*, which is organized as a circular buffer. Each memory updates log record has four fields: `read_addr`, `write_addr`, `len`, and `data`. The address space of a program can be modified by either the program itself or by a library function call made by the program. To handle the updates of the former type, *APR* logs the effects of assignment statements of the following form:  $X = Y$ , where  $X$  and  $Y$  are directly referenced variables, array references (e.g.,  $a[i]$ ) or de-referenced variables (e.g.,  $*(a+1)$ ). The *read address* field contains the address of the right-hand-side variable of the assignment operation, in this case  $Y$ 's address. The *write address* field holds the address of the left-hand-side variable being modified, in this case  $X$ 's address. The *length* field is the size of the modified variable, size of  $X$  in this case. The *data* field stores the pre-image of  $X$ , the variable being written to. It is not always possible to uniquely identify the read address of an assignment operation, for instance if  $Y$  is a complex expression containing a number of variables or a function call. In this case the read address is set to “-1,” which indicates that the data origin of this assignment is unknown.

For damage repair, knowledge of the origin of a global/static variable is not essential, and therefore the field *read address* is redundant. However, it can be used to compute the backward dynamic slice for a particular memory location, that is, to find all the other memory locations whose values influence the value of that location. The dynamic slice is useful in determining the scope of repair after a buffer-overflow attack. The above memory updates logging algorithm implements both state checkpointing and data dependency tracking. Moreover, the *APR* compiler inserts logging code for each assignment operation of the form specified above without performing any sophisticated data or control flow analysis. As a result, the implementation complexity of the *APR* compiler is greatly simplified.

To reduce the memory updates logging overhead, *APR* tries to avoid unnecessary logging operations as much as possible. In its default mode, *APR* chooses not to log updates of the form  $X=Y$  where  $X$  is a local variable referenced directly, based on the assumption that local variables referenced directly are usually used as temp variables (for example, as loop variables) and are not related to global/static variables.

Function class	Libc functions
Copying/concatenation	memcpy(), mempcpy(), memmove(), strcpy(), strncpy(), strcat(), strncat(), bcopy()
Network I/O	readv(), recv(), recvfrom()
Inter-procedural jumps	setjmp(), longjmp()
Memory management	malloc(), calloc(), realloc(), free(), strdup()
Privilege management	seteuid(), setreuid(), setegid(), setregid()
Process creation	fork()
File I/O	read(), fread(), scanf(), vscanf(), fscanf(), vfscanf(), gets(), fgets()
Format string	sprintf(), snprintf(), vsprintf(), vsnprintf(),

Table 1: The set of library functions that *APR* needs to proxy to support memory updates logging.

*APR*'s memory updates log contains additional information such as marks that indicate function boundaries. We call such records *tags*. There are several types of tags, for example a *function entry tag* and a *function exit tag*. The tag type is stored in field `read_addr`. The remaining fields are used differently for each tag. We will describe each tag type one by one in Section 5.3. The code required to insert tags is automatically added by the *APR* compiler.

To log memory state updates performed by standard *libc* library functions such as `memcpy()`, *APR* needs to proxy some of them to capture the updates. The complete list of proxied functions is presented in Table 1. Below we will consider each group of functions in more detail.

**String Copying and Concatenating Functions.** Each proxy function from this group generates a log record. For instance, a log record for a `strcpy(a, b)` function call contains the address of `b` in its `read_addr` field, the address of `a` in its `write_addr` field, `strlen(b)` in its `len` field. The `data` field contains the pre-image of buffer `a` of length `strlen(b)`. After generating a log record the proxy function calls the corresponding *libc* function and returns its result.

**Network and File Input Functions.** These proxy functions generate two log records each time they are called. The `read_addr` field of both records is set to a special value indicating the external source of the data being logged. The `data` field of the first log record stores the pre-image of the memory buffer. The `data` field of the second record stores the post-image of the memory buffer, that is, the data that was actually read from the network or a file. The latter record is used for attack identification purposes as described in Section 5.2.

**Memory Management Functions.** Each function in this group is proxied because it modifies the heap memory pool and these updates need to be undone at repair time. The `proxy_malloc()` calls `malloc()` first and stores the address of the newly allocated object in the memory updates log. At repair time if this record needs to be rolled back, this memory object is freed.

At repair time we also need to reallocate objects that were previously deallocated. This is achieved by proxying `free()`. A straightforward way to restore the object that was deallocated is to allocate it again with `malloc()`. However, the new

object may be created at a different memory location and all earlier references to it in the memory updates log will need to be remapped. Instead, we use a *deferred free()* approach. When the program calls `free()`, the `proxy_free()` function just puts the address of the object into the log without freeing up the object. At repair time, we do not need to do anything to restore the original object since it is kept in the memory.

Finally, the `proxy_realloc()` function saves the original pointer in the buffer, replaces the original `realloc()` call with a `malloc()` call and saves the pointer to the newly allocated memory as well. Then it copies the data to the newly allocated buffer. The length of the data being copied is obtained from the memory buffer header that is preceding the data itself. At repair time, the newly allocated object is deallocated.

Because the capacity of the memory updates log is limited, its records are used in a circular fashion. That is, they will be reused if the program runs long enough. When an undo log record for an operation is reused, its previous content is cleared, and it is no longer possible to undo that operation. When a log entry associated with `proxy_free()` is reused, it executes the corresponding `free()` operation that was deferred before.

**Inter-Procedural Jump Functions.** Function `longjmp()` performs an inter-procedural jump to one of the dynamic ancestors of the current function. To keep the memory updates log consistent, we need to add a proper number of function exit tags to the log. In theory this number should be equal to the number of functions that `longjmp()` skips. To determine this value at run time, *APR* proxies both `setjmp(jmp_buf)` and `longjmp(jmp_buf, state)`. The `proxy_setjmp()` function logs the address of the `jmp_buf` variable. The `proxy_longjmp()` function searches the memory updates log for a log entry corresponding to a `setjmp()` call that filled in the `jmp_buf` variable used in `proxy_longjmp()`. Once locating the proper log entry, *APR* can then find out the nesting level of the current function with respect to the target function and thus compute the required number of function exit tags to be added.

**Privilege Management Functions.** Many programs change their effective user ID and group ID values for security reasons. At repair time, the proper values need to be restored so that the program has the same access rights as it had before the failure. This is achieved by proxying functions such as `seteuid()` and `setegid()`. These functions save the original value of uid or gid in the `data` field of a memory updates log record. However, when a privileged process calls `setuid()` to replace its effective user ID with a nonzero effective user ID, it is impossible to restore the old effective user ID without special operating system support.

**Process Management Functions.** When a program compiled by *APR* forks a new process, the two processes can access their memory updates logs concurrently because of the copy-on-write semantics of the `fork()` system call. In this case, two versions of the log are created automatically by the OS. At repair time, both parent and child processes need to be repaired. The current version of *APR* does not consider the problem of cascading rollback. That is, if a failure was detected in the parent process then all child processes that have been forked after the new restart point are killed. However, if a failure is detected in the child process and the restart point is chosen to be before the point where it was forked, the process is terminated without having any effect on the parent process. In the `proxy_fork()` function, *APR* inserts special tags into the parent's and child's process logs to facilitate the repair.

Some applications provide an alternative implementation of standard *libc* functions. *APR* adds memory state logging code to these functions during compilation, as long as they are written in standard C. If, however, the new *libc* functions are

implemented using inline assembly, then the current *APR* prototype cannot instrument them.

## 4 Undo Support for Interactive Applications

Many interactive editing applications such as WORD and PowerPoint support an undo option. In most cases, programmers have to explicitly implement this feature. The *APR* compiler frees the programmer from the burden of developing undo code by automatically augmenting these programs with logging and repair logic.

An interactive application is typically structured as a main event loop that receives user commands, e.g., insert a string or draw a rectangle, interprets them and updates some internal data structures, and draws the effects on the screen. Occasionally, it dumps the internal data structures to a on-disk file either on user demand or periodically.

The *APR* compiler provides an `undo()` function that allows the programmer to undo a user command dispatched from the main processing loop. When this undo function is invoked, it undoes the effects of the non-undo user command immediately preceding it. After the effects of a previous user command are undone, the program's execution resumes from the point where the undo function was called, and then to the main processing loop waiting for the next user command. Repeated invocations of the undo command will successively erase the effects of earlier non-undo user commands in reverse order.

*APR* also provides a special function called `mark()` in its API for a programmer to specify the scope of the undo command. Typically, the programmer calls the `mark()` function before and after an undoable user command is executed, and these two calls insert special tags into the memory updates log to mark the beginning and the end of the scope of the undoable operation. In addition, the *APR* compiler supports another function called `flip_log_flag()` that turns logging on and off. By turning logging off temporarily when it is not required (for example, in the GUI part of the program), the programmer can reduce the log space required to support undo command.

To reduce logging run-time overhead and log space consumption, the *APR* compiler does not keep track of memory updates that are related to the screen display. These updates do not need to be logged because they are completely derived from the application's internal data structures, and will be recomputed once the internal data structures are updated. The current *APR* prototype cannot distinguish between normal memory updates and screen-related memory updates, and thus requires programmers to explicitly use function `flip_log_flag()` where logging is not necessary.

*APR* needs to support undo of both memory state and file system state for interactive applications. For example, if the user saves a file to the disk and then decides to undo a couple of commands, then the original version of the saved file needs to be restored. *APR* uses a lightweight file state updates tracking mechanism specifically designed for interactive applications. The disk space overhead of this approach is less than 100%, which means that it stores at most one additional copy of the original file, no matter how many times the file is saved. The easiest way to implement file system state undo is to log every file I/O operation. However, the overhead of such an approach is very high. Instead, the *APR* compiler exploits the fact that typical interactive applications have only two file I/O functions: `readD()` for reading in a document and `writed()` for writing the current document to the disk, and they are called from the main processing loop when the user invokes a command such as "open a file" or "save a file". We assume that function `writed()` does not perform any

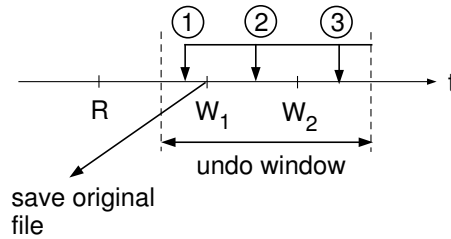


Figure 1: File state updates logging. A file was first read (R), then written twice (W1 and W2). The three restart points, labeled as 1, 2, and 3, correspond to the three cases of the file state repair algorithm.

partial writes or appends to existing files but rather always saves the entire file. Another assumption that we make is that `readD()` and `writeD()` are the only functions that perform file I/O. Therefore, *APR* only logs invocations of `writeD()` rather than calls to the underlying file I/O *libc* functions. When `writeD()` is called, a record with tag `RAD_FILE_WRITE` is inserted to the log.

Given the undo window operations beyond which can no longer be undone, *APR* maintains two versions of each file updated by an interactive application, the most up-to-date version (called `FO`), and the most recent version before the undo window (called `FT`). When a file is saved the first time, the original version is copied to the `/tmp` directory (`FT`), and all updates are made to the current version (`FO`). This continues until a log record with tag `RAD_FILE_WRITE` is recycled. At that point, `FT` needs to be updated so that it can incorporate the effect of the associated `writeD()`.

Figure 1 illustrates the file state updates logging algorithm. In the beginning, file `FT` is created from the original version of the file after the first write `W1` and is not affected by `W2`. Only when `W1` falls out of the undo window will `FT` need to be modified. When an undo command is invoked, `FO` may need to be modified, depending on whether there are intermediate `writeD()` operations. There are three cases:

1. When there is a `writeD()` mark only on the right hand side of the new restart point, move `FT` to `FO`.
2. When there are `writeD()` marks on both sides of the new restart point, rollback to the `writeD()` mark on the left hand side of the new restart point, call `writeD()`, and roll forward to the original restart point. Here we use the assumption that `writeD()` always saves the entire file. Therefore, the correct version of `FO` can be created by restoring the underlying memory state and calling `writeD()`.
3. When there is a `writeD()` mark only on the left hand side of the new restart point or no `writeD()` marks in the current undo window at all, `FO` does not need to be changed.

## 5 Undo Support for Network Applications

The most common way to hijack control of a remote system is to exploit a buffer overflow vulnerability in a network application. In order to redirect the control flow to the injected code, the attacker needs to overwrite some control-sensitive

data structure such as a return address in the target program. When the compromised control-sensitive data structure is used, the control is transferred to the attacker's code. Without any special support, the program will not be able to distinguish an illegitimate control pointer from a legitimate one.

The *APR* compiler can automatically detect a buffer overflow attack, repair it, and further identify the packet that is responsible for the attack. To repair the effect of the malicious network packet(s) that led to the buffer overflow attack, *APR* generates code to identify the point in the memory updates log at which the packet was read in and then undo all the records from that point to the end. As a result, the attack packet is also automatically identified. After an attacked application is repaired, it should not resume from the point immediately after the attack is detected. Instead, it should resume from the point before the attack packet is received. The *APR* compiler uses a sophisticated procedure to find the new restart point, which is to be described later.

## 5.1 Attack Detection

Most of the control-hijacking attacks modify some control-sensitive data structures in the victim program, such as a return address, a function pointer, or a jump table, through buffer overflowing. Once the compromised data structure is used in a control transfer, the attacker hijacks the control of the application.

The approach to attack detection used by *APR* is similar to that developed in the RAD project [12]. To detect control-hijacking attacks at run time, the *APR* compiler maintains the original image of every control-sensitive data structure, and at the time of the control transfer compares the current value of the associated control-sensitive data structure with its original image to determine whether it has been modified via buffer overflowing. The current *APR* prototype protects only return addresses and function pointers as they are the most common attack targets. In particular, the *APR* compiler instruments an input program as follows. (1) At the function prologue, the return address is stored in the return address buffer. At the function epilogue, the return address on the stack is compared with the stored value in the return address buffer. If there is a mismatch, the return address has been tampered with and a control-hijacking attack is detected. (2) Every time a function pointer is modified in the program, its newest value is stored in an existing or a new entry of the function pointer buffer. This includes the case when a function pointer is passed as an input argument into a function. Each entry of the function pointer buffer contains two fields: the address of a function pointer variable and its value. Every time a function pointer is about to be used to make a function call, its current value is checked against the function pointer's stored value. Mismatch of these two values indicates that an attack is taking place.

Because the return address buffer and the function pointer buffer are supposed to contain the ground truth, they should be well protected so that tampering via buffer overflowing is impossible. Otherwise, if an attacker can overflow both a control-sensitive data structure and its associated duplicate buffer, she can defeat this attack detection method. Towards this end, both the return address buffer and the function pointer buffer are sandwiched inside a pair of read-only pages. Any attempts to modify these two buffers via overflowing will result in protection faults.

In theory, the *APR* compiler can also protect jump tables in the same way as function pointers. However, because there have never been any real control-hijacking attacks that tamper with jump tables, for simplicity we chose to ignore jump table protection in the current prototype.

```

cur_addr=MA;
while (more_log_entries && cur_addr≠0)
    ent=get_prev_log_entry();
    if ent.write_addr ≤cur_addr && ent.write_addr+ent.len>cur_addr
        then cur_addr=ent.read_addr+(cur_addr-ent.write_addr);
    end;
if (cur_addr≠0)
    { printf("Can't find source of attack\n"); exit(0); }
/* ent is the required log entry */

```

Figure 2: The traceback algorithm used to locate the source of a buffer overflow attack based on a corrupted control-sensitive data structure.

## 5.2 Attack Identification

Upon detecting a control-hijacking attack, we assume that the corrupted control-sensitive data structure is compromised by some data that might have been read from the console by a `gets()` call or from a network socket by a `recv()` call. In these cases, it is important to identify the source of corruption to determine the scope of repair and to prevent the same compromise from happening again. This data can be sent to a front-end intrusion-detection system, which can then use it to prevent the same attack from reaching internal hosts again. This automatic attack packets extraction capability protects an enterprise from worm-like attacks, where attacking or compromised hosts tend to send out attack packets that are largely the same. Of course, it is also possible that the control-sensitive data structure was actually overwritten due to a mistake in the program's internal logic. In this case, the program should be just terminated since no automatic repairing can stop the same compromise from recurring.

To identify the data item read from the network or a file that is responsible for the corruption of a control-sensitive data structure, we need to trace back the dependency graph, starting from the corrupted control-sensitive data structure. This tracing relies on the read address and write address fields of the memory updates log entries. Let *MA* (modified address) be the address of a corrupted control-sensitive data structure, such as a return address or a function pointer. In most cases, it was tampered with as a result of an unchecked array-to-array copy operation such as `strcpy()`. Each of such modifications leaves a record in the memory updates log. Therefore, the tracing begins with the most recent memory updates log entry whose write address is equal to *MA*, and uses the read address field of this entry as a key to search the memory updates log to find the most recent log entry whose write address matches it, etc. This process continues iteratively until reaching a memory updates log entry whose read address is set to one of the special values described below, which means that the data written to the write address of that entry comes from an external source. The above trace-back algorithm is formally described in Figure 2.

### 5.3 Attack Repair

There are two issues involved in this program state repair process: (1) From which state should a victim program restart? (2) How to restart a victim program without special OS support?

Because *APR* logs only updates to global, static, and array-like variables, it can only restart a program from the entry point of a function. Logically, the function to resume from (called `f_restart`) should be the *least common ancestor* of the function in which the attack was detected (`f_attack`) and the function in which the malicious external data was read in (`f_read`), because the stack frame of the dynamic parent of `f_restart` has not changed as the control goes from `f_read` to `f_attack`. Because *APR* does not log any local variable updates, it is not possible to bring the program back to a consistent state older than the state in which it was right before `f_restart` was called. There is an exception from this rule, however. If there are no local variable updates in `f_restart` between the point after `f_read` returns and before `f_attack` begins, we can safely restart the execution from `f_read` instead of `f_restart`.

Sometimes, it may be the case that the whole program needs to be restarted from the beginning. Indeed, this happens if `f_restart` turns out to be function `main()` and there are some local variable updates made between when `f_read` returns and `f_attack` begins. While evaluating *APR* we have encountered one such program. One way to avoid this problem is to track all variable updates including local ones, but that may significantly increase the run-time overhead. If the repair algorithm finds that the program needs to be restarted from the beginning, the program is simply terminated and restarted afterwards.

Figure 3 illustrates how the restart point is chosen with a typical buffer overflow attack scenario. In this case, either `f2()` or `f1()` can be chosen as the new restart point. The decision depends on whether there are any local variable updates in `f1()` after return from `f2()` until the call to `f4()`. *APR* does not require any system support for program restart. Instead, it uses inter-procedural jump functions `setjmp()` and `longjmp()` to implement the resume functionality.

Figure 4 shows the algorithm that *APR* uses to find the least common dynamic ancestor between `f_read` and `f_attack`, from the memory updates log. Logically, the algorithm traverses the memory updates log backwards to find the first function whose function entry tag is earlier than the function entry tag of both functions.

Finding a restart point requires augmentation of the memory updates log with several types of tags. These tags are *function entry tag*, *function exit tag*, *jump buffer tag*, and *first local update tag*. Upon entering a function *APR* inserts a function entry tag into the memory updates log. Similarly, when the function returns a function exit tag is inserted. When a function call is made, *APR* inserts a call to `setjmp(buf)` where `buf` is the `data` field of a memory updates log record. The `read_addr` of this record is set to the jump buffer tag. This marks the point preceding the function call a potential restart point. At repair time, the control can be transferred to this point by performing `longjmp(buf)`. Finally, the first local update tag is inserted to the log when the first update to a local variable is encountered after a function call. These tags are used at repair time to determine the actual restart point, which can be either `f_read` if no such tags are found in `f_restart` between the call to `f_read` and the call to `f_attack`, or `f_restart` if at least one local update tag was found.

Once the restart point is determined, the memory state of the program needs to be rolled back to the state corresponding to the new execution point. To do so, the repair module needs to traverse the memory updates log in the reverse direction

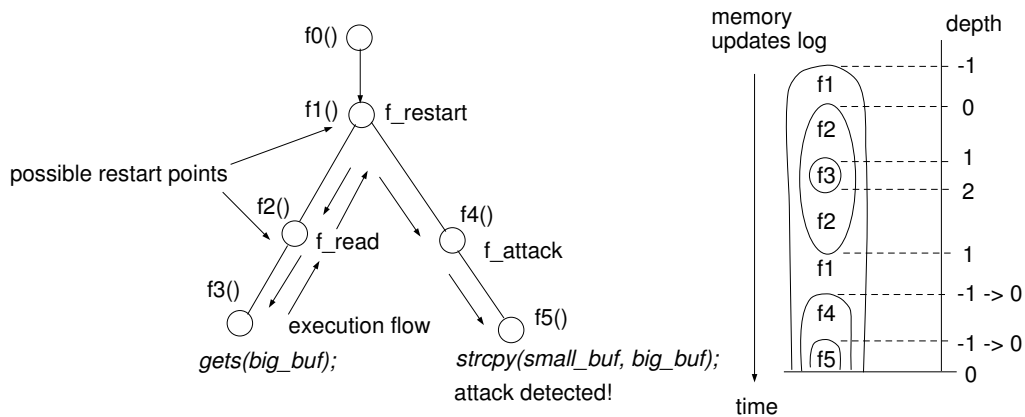


Figure 3: An example illustrating how to identify the least common dynamic ancestor in the function call graph and use it as the restart point. The right-hand side shows the memory updates log where f1–f5 are the same functions as those on the left-hand side. The ovals correspond to function boundaries. The depth values are the values of depth variable that is defined in Figure 4.

```

f_read — function in which malicious data was read in;
ent_beg=first log entry of f_read (function entry tag);
depth=0;
ent=last_log_entry();
while (ent!=ent_beg)
    if (ent.tag is function entry tag) then depth--;
    if (ent.tag is function exit tag) then depth++;
    if (depth<0) then depth=0;
    ent=get_prev_log_entry();
end;
/* second phase */
while (depth≥0)
    if (ent.tag is function entry tag) then depth--;
    if (ent.tag is function exit tag) then depth++;
    ent=get_prev_log_entry();
end;

```

Figure 4: Algorithm for finding the least common dynamic ancestor in the function call graph.

Program	Type	Client request/user action	No. of Repetitions
pico	text editor	type a character	—
figurine	graphics editor	move a polygon	—
dhex	binary editor	change a byte	—
ghttpd	HTTP server	fetch a 10KB HTML page	5,000
drcatd	remote cat daemon	fetch a 10KB file	1,000
named	DNS server	lookup of a domain name	10,000
qpopper	POP3 server	fetch a 1 KB message	200
proftpd	FTP server	fetch a 40 KB file	100

Table 2: Test programs and corresponding performance tests.

until it reaches the restart point, undoing each global variable update along the way. After the undo, the recovery module performs a `longjmp()` using the `jmp_buf` corresponding to the restart point.

## 6 Evaluation of APR

### 6.1 Experimental Setup

In this section we evaluate *APR* in terms of its compile-time and run-time characteristics such as the number of instrumented lines of the source code, the amount of log information generated at run-time, and the performance overhead of the instrumented program. We also describe our experiences in applying the *APR* compiler to several network daemons and interactive applications.

We used a suite of five network daemons and three interactive applications in our experiments. The following network daemons were in our test suite: `ghttpd` 1.4 — an http server, `drcatd` 0.5.0 — a remote cat daemon, `named` 8.1 — DNS daemon which is a part of BIND program, `qpopper` 4.0.4 — a POP3 server, and `proftpd` 1.2.9 — an FTP server. The interactive applications were: `pico` 4.61 — a text editor which is a part of the widespread `pine` mail editor, `figurine` 1.0.5 — a vector graphics editor, and `dhex` 0.54 — a hexadecimal binary editor. In order to test the intrusion-resilience of the instrumented network daemons, we used several exploit programs available at Fyodor’s Remote Exploit Archive [14] and Securiteam’s website [4].

The hardware setup used in the performance experiments is as follows. The network daemon being tested was running on a server machine with a Pentium-4M 1.7GHz processor and 512 MB of RAM. There were two client machines with AMD Athlon 1.7GHz processors equipped with 512 MB of RAM each. The interactive applications were compiled and run on the server machine. All machines were running the Linux 2.4.19 kernel. The machines were located in the same 100 Mbps local network. The programs were compiled on the server machine with options `-g -O`.

To measure the performance characteristics of the instrumented network daemons, the client machines were running special programs that were simultaneously sending a number of requests to the server machine. In order to test the interactive

Program	Original undo (LOC)	APR undo (LOC)	Saved LOC (%)	Instrumented by APR (LOC)
pico	N/A	16	N/A	2563
figurine	650	35	94.6	5123
dhex	25	12	52.0	589

Table 3: Number of programmer-written lines of code (LOC) required to support undo with and without APR as well as the number of automatically instrumented statements.

applications, a testing user was performing a number of standard actions. Table 2 summarizes the client requests and the user actions that were used to evaluate the performance of the instrumented programs.

## 6.2 Performance Measurements for Instrumented Interactive Programs

For interactive applications, *APR* either added an undo command if it did not exist in the original version of the program or improved the precision of the undo command if it existed before instrumentation. The goal of this series of experiments is three-fold. First, we aim to quantify the amount of work that *APR* can save for an application programmer. This amount can be expressed as the ratio between the number of lines of the original undo code written by programmers (if it exists) and the number of lines of undo code written by programmers when *APR* is used. In addition, we also measure the number of the statements automatically instrumented by *APR*. Second, we measure the amount of log information generated by a single user action, which allows one to estimate the required log size for a given undo window. Finally we also want to measure the programming efforts required to apply *APR* to an existing interactive application as well as the impact of the new undo command on application usability.

Table 3 shows that using *APR* can indeed save a significant amount of programmer’s effort. The number of programmer-written lines of the undo code when *APR* is used is less than 40 for all the programs that we have studied. Moreover, the structure of this code is the same for all applications: one needs to make a call to the repair API that *APR* provides to restore the underlying file system and memory states first and then call a proper function from the GUI part of the application to redraw the screen content. Therefore, it is easy for a programmer to add *APR* support to a new application if she has done this previously for other applications. Another interesting observation is the fact that the number of automatically instrumented lines of code is much higher than the number of programmer-written lines of undo code. This happens because the *APR* compiler makes a conservative assumption about each memory state update. That is, each memory update is logged even though the same memory location can be changed several times later on. On the other hand, the code written by the human programmers usually takes snapshots of the final versions of the necessary data structures.

However, because of its comprehensive nature the automatically generated undo code allows users to undo commands that could not be undone previously by the programmer-written code. For example, the polyline command in *figurine* editor allows the user to draw a multi-segment line and can be undone as a whole only after the line drawing is finished. With *APR* support, however, it becomes possible to return to the line drawing state and to continue drawing the line. The undo command in the *dhex* editor allows one to undo only the changes to the file being edited. With *APR* support, however,

Program	Log records	APR undo size, bytes	Original undo size, bytes
pico	22	61	N/A
figurine	1187	2984	324
dhex	20	66	10

Table 4: Number of log records and the amount of undo information (in bytes) generated for a single user action by the APR-compiled code as well as the amount of undo information (in bytes) per user action generated by the original human-written undo code.

it becomes possible to undo cursor movements as well. This enhancement adds a “bookmark” functionality to the editor, that is it allows the user to return to a previously visited point after she moves the cursor to another point of the file.

Table 4 presents the amount of undo information generated by the interactive programs for a single user action as well as the amount of undo information stored by the original human-written undo logic of the program. Although this amount depends on the exact command being executed, we believe that the user actions listed in Table 2 are among the most typical user actions. The results indicate the the automatically generated undo code logs 7-9 times as much information as is actually required to support undo. Therefore, the current checkpointing algorithm still has a lot of space for improvement. These results suggest that a memory updates log containing 50,000 records would be sufficient to implement a 50 step undo for `figurine` and almost infinite undo for `pico` and `dhex`. In addition to the `data` field that stores the application data, a log record has three other fields which occupy 12 bytes in total. Taking this into account, the memory updates log consisting of 50,000 records would require about 850 KB of memory.

### 6.3 Performance Measurements for Instrumented Network Daemons

We compiled the chosen network daemon programs using *APR*, and conducted a series of experiments to evaluate their correctness and measure their performance overhead. First, we measured the number of memory updates log records as well as the total size of memory updates log (in KB) for a single client request as described in Table 2. The results presented in Table 5 suggest that the correlation between the number of log records and the actual amount of data written to the log is non-linear, because different log records have different actual size. For example, the size of a single variable update log record is 16 bytes (4 bytes for read address, 4 bytes for write address, 4 bytes for data length and 4 bytes the actual payload) whereas the size of a log record generated by proxied string manipulation and network or file I/O functions can vary from several bytes to 1 KB.

Then we measured the run-time performance overhead of the instrumented programs, which is considered the most important metric of the *APR* compiler. In each test, we repeated the same user request a number of times, as described in Table 2. The measurements from these experiments are presented in Figure 5 and suggest that the run-time overhead can vary significantly depending on the programs’ memory access behavior and can range from 8% to 60%. In particular, the programming style in which the program was written matters significantly. For example, certain programming techniques tend to increase the run-time overhead, such as breaking up the program into a large number of small functions and using

Program	Log records	Log size, KB
ghttpd	457	32
drcatd	4,000	408
named	832	39
qpopper	27,000	586
proftpd	70,000	2073

Table 5: Number of log records and log size (in kilobytes) generated for a single client request as described in Table 2.

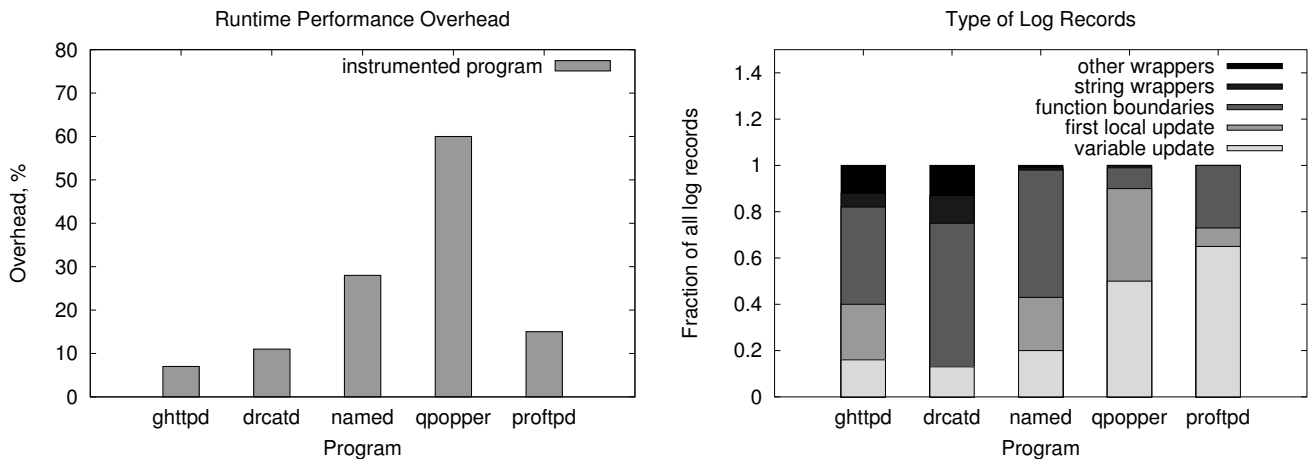


Figure 5: Run-time overheads for different modes of compilation (left) and the relative frequency of different types of log records (right).

pointer arithmetic extensively. These results also suggest a need for a more intelligent checkpointing mechanism that can help reduce the overhead. For example, instead of saving the data pre-image upon each update, one can save the pre-image of the whole data structure once upon function start.

We have also measured the relative frequency of each type of log records written to the memory updates log during a performance benchmark run. The results are presented in Figure 5. Although there seems to be no direct correlation between the frequency of types of log records and the run-time overhead, the results show that programs with higher overhead (such as `qpopper`) insert more records for variable updates and first local update tags. Indeed, these two types of records account for 90% of all records that `qpopper` had written to the log. These results suggest once again a need for a more sophisticated checkpointing algorithm that can help reduce the number of variable update log records as well as a more intelligent mechanism for choosing the restart points. The current *APR* prototype relies on information about local updates to determine where restart points are, and consequently generates much more first local update log records than necessary. Ideally, dependency analysis techniques such as slicing can identify points in the program that can eventually lead to a function that reads external data, and thus points that can be potential restart points.

In the next series of experiments we measured the amount of file and network I/O activity during the execution of the

Program	File IN	File OUT	Net IN	Net OUT
ghttpd	45	0	1	49
drcatd	319	0	3	320
named	0	0	1	1
qpopper	41	80	5	7
proftpd	13	63	11	61

Table 6: Network and file I/O activity for a single client request as described in Table 2.

programs in our test suite. This information can help answer the question of whether the file system and network undo is indeed required for the repair process or the programs can be repaired and continue their execution without file system and network undo. The results are presented in Table 6. The results showed that 3 out of 5 programs that we tested do not perform any file output operations when serving a single client request. Our analysis of the source code of the remaining two programs showed that the file output operations performed by those programs are used to create temp files and write logging information. We believe that this information is not a critical part of program’s state and therefore leaving it after an attack will not bring the program into an inconsistent state. The network output operations performed by the programs are related to communicating with the client that initiated the connection only. Therefore, if that client turns out to be malicious there is no need to undo the effects of network operations for such a client. These observations suggest that file system and network undo support are not be required for network daemons we have studied.

We used exploit code from public databases to compromise `named`, `drcatd`, and `ghttpd`. In all tests, the attack was detected and identified successfully. For `named` and `drcatd`, the repair procedure was able to fix the programs and let them continue normal execution. For other programs, they have to be restarted because the repair procedure decided that `f_restart` was `main()`, and in addition there were a number of local variable updates between `f_read` and `f_attack`. There are two ways to avoid restarting a compromised network application. The first is to reorganize the source code manually by putting potentially vulnerable parts of the code into a separate function so that the execution can be restarted from it in case of attack. However, this solution requires some understanding of the source code of the program and therefore is not suitable for automatic program protection. The second solution is to log all memory updates including local ones. However, the current version of *APR* cannot tell automatically whether tracking global updates only will be sufficient or not. This option can be turned on and off manually. When compiled with this option turned on `drcatd` can repair itself and continue normal execution.

## 7 Conclusion

In this paper we presented the first known compiler that can automatically add code to an arbitrary C program to undo undesirable side effects in the form of updates to the address space and file system. This code can be used to repair programs compromised by buffer-overflow attacks or to support undo operations in an interactive application. In the former

case, no human intervention is required. In the latter case, the programming efforts spent on using *APR* are much less than efforts required to manually codify the undo logic. In addition, the performance overhead of the *APR* transformations is shown to be quite modest, even without any aggressive optimizations.

There are a number of ways in which the *APR* prototype can be improved. First, we aim to improve the efficiency of the memory updates logging mechanism by employing control flow analysis. Currently, *APR* logs every update to each global variable, even though in theory only the first one needs to be logged. Another problem with the current logging mechanism is that it may miss certain data dependencies, for example, when a local variable is used to transfer information between two global variables. Detecting all data dependencies is important for successful exploit identification. Comprehensive data dependency analysis is required to improve the accuracy of attack identification. We are also going to address multi-threading issues in more detail in the next version of *APR*. Multiple threads of the same program can concurrently access the memory updates log and other global data structures, and thus introduce additional data dependencies. At repair time, *APR* needs to determine which threads should be rolled back, restore the state of each such thread to the corresponding pre-attack state, and resume its execution.

## References

- [1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution-backtracking approach to debugging. In *IEEE Software*, May 1981.
- [2] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 15(5), 2002.
- [3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.
- [4] Beyond Security's SecuriTeam. <http://www.securiteam.com>.
- [5] A. Brown and D. A. Patterson. Embracing failure: A case for recovery-oriented computing (ROC). In *Proceedings of 2001 High Performance Transaction Processing Symposium*, 2001.
- [6] A. Brown and D. A. Patterson. To err is human. In *Proceedings of the First Workshop on evaluating and architecting system dependability (EASY'01)*, 2001.
- [7] A. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to dependability. In *Proceedings of the 10th ACM SIGOPS European Workshop*, 2002.
- [8] A. Brown and D. A. Patterson. Undo for operators: building an undoable e-mail store. In *Proceedings of the 2003 USENIX Annual Technical Conference*, 2003.
- [9] G. Candea and A. Fox. Crash-only software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, 2003.

- [10] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. A microrebootable system — design, implementation, and evaluation. In *Proceedings of Operating System Design and Implementation Conference*, 2004.
- [11] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), 1994.
- [12] T-C. Chiueh and F-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proc. of 21st Intl. Conf. on Distributed Computing Systems*, 2001.
- [13] S. Feldman and C. Brown. Igor: A system for program debugging via reversible execution. In *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, January 1989.
- [14] Fyodor. Remote exploits. [http://www.insecure.org/spl0its\\_remote.html](http://www.insecure.org/spl0its_remote.html).
- [15] M. Kifer, P. Lewis, and A. Bernstein. *Database and transaction processing: an application-oriented approach*. Addison-Wesley, 2002.
- [16] P. Liu. Architectures for intrusion tolerant database systems. In *Proceedings of Annual Computer Security Applications Conference*, 2002.
- [17] P. Liu. Itdb: an attack self-healing database system prototype. In *Proceedings of DARPA Information Survivability Conference and Exposition*, 2003.
- [18] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, January 1989.
- [19] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. In *UC Berkeley Computer Science Technical Report UCB/CSD-02-1175*, 2002.
- [20] A. Smirnov and T-C. Chiueh. A portable implementation framework for intrusion-resilient database management systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2004.
- [21] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of USENIX Annual Technical Conference*, June 2000.
- [22] J. Strunk, G. Goodson, M. Scheinholtz, Craig Soules, and Gregory Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of Operating Systems Design and Implementation Conference*, 2000.
- [23] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote, and P. Khosla. Survivable information storage systems. *IEEE Computer*, 2(1), 2000.
- [24] N. Zhu and T-C. Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2003.