Software Fault Isolation: a first step towards Microreboot in legacy C applications

University of Maryland Institute for Advanced Computer Studies
Technical Report 2006-31
Timothy Fraser (*tfraser@umiacs.umd.edu*)

20 May 2005

Abstract

Microreboot is an attractive technique for recovering an application after a non-malicious failure or deliberate integrity breach even in cases where the precise cause of the failure or breach are not known. Unfortunately, Microreboot functionality has so far been demonstrated only with Java applications meeting a set of peculiar Crash-only architectural requirements. This report describes a method of using Software Fault Isolation techniques to meet some of these architectural requirements in C programs, thereby taking a first step towards making Microreboot available for retrofit in legacy C applications.

1 Introduction

Candea and others have developed the notion of Microreboot—a technique for restoring failed applications to a working state [4]. Microreboot works by restoring part of a failed application to its initial state while keeping the balance of the state as it was before the failure. If failures persist after an initial Microreboot, the technique can be applied repeatedly, resetting more and more applications state each time, until the application resumes healthy operation. This technique has a number of admirable features: it presents the possibility of losing less valuable state than full reboots, and like full reboots it does not require knowledge of what actually caused the failure.

Unfortunately, Candea's original notion of Microreboot can be applied only to applications based on an unusual "crash-only" architecture [3]. Such software is made up of many small well-isolated components that store their important state in external stable repositories such as transaction-oriented databases, communicate using retryable requests, and manage resources sharing with expirable leases. Although Candea's experiments with a specially-constructed Java-based crash-only e-commerce test application showed good results, the architecture of this application is so unusual it is difficult to imagine how Microreboot might be applied to many existing real-world applications. These unusual architectural requirements are inconvenient, as Microreboot could otherwise be used to help recover the world's many legacy C applications from from non-malicious failures and deliberate compromises.

The Rio File Cache work of Chen and others provides a hint as to how at least one of the odd requirements of crash-only software might be achieved via retrofit in legacy C applications [5]—specifically, the requirement that critical application state be stored in a manner that enables it to survive application restarts. In Crash-only applications, this requirement is met by storing critical state in external transactional databases.

Using Software Fault Isolation (SFI) techniques [22], Chen retrofitted a legacy Digital Unix kernel with the ability to maintain the contents of its file cache across warm reboots across warm reboots. What Candea achieved by forcing crash-only applications to keep their state in external transactional databases, Chen achieved (mainly) by using SFI to avoid the erasure or unintentional modification of the file cache, even as the operating system was in its death throes.

This report describes the results of a small effort to investigate the use of Rio-File-Cache-like SFI retrofit techniques in legacy C applications in order to maintain the contents of critical in-memory application data structures across software restarts. A software restart is the application equivalent of an operating system warm reboot: the failed application's process does not terminate upon receiving a failure signal (SIGSEGV, SIGBUS, and so on). Instead it catches the signal, unwinds its stack and jumps back to the beginning of the program.

The core of this effort is the experimental implementation of the Rio File Cache SFI technique in a small C application. The primary result of this experiment is evidence indicating that the Rio File Cache SFI technique can indeed be used to provide state preservation functionality equivalent to the Crash-only architecture's use of exter-

nal transactional databases, but only when coupled with additional functionality to ensure the consistency of the preserved state in the presence of restarts that unexpectedly catch the application in mid-update. In the course of the experimental implementation, dealing with this consistency issue was at least an interesting as implementing SFI. Doing so lead to a secondary result: potential insights into the nature of "semantic integrity"—a topic of practical interest to the implementors of run-time integrity monitors such as Copilot [20], Backdoors [1], and VMI [11].

The remainder of this report is organized as follows: section 2 describes the Crash-only architectural requirements required to support Microreboot functionality in greater detail, and states precisely which of these requirements the experimental use of SFI is meant to meet in legacy C applications. Section 3 briefly summarizes the Rio File Cache approach used as the basis for this experiment. Section 4 describes the experimental implementation of SFI in a simple C application. Section 5 explains how this experimental implementation made the need for an consistency-assuring mechanism in addition to SFI clear, and describes how a suitable solution was ultimately chosen from a set of alternatives. Section 6 describes work related to this effort. Section 7 presents conclusions. Section 8 describes the potential insights into semantic integrity resulting from dealing with state consistency issues in the experimental implementation and outlines some avenues for future work.

2 Microreboot

The Microreboot work of Candea and others [4] demonstrates a method of restoring individual unhealthy components of a system to a known good state without causing conflicts with the remainder of the system. Like traditional reboots, this technique is appealing because it can conceivably restore an application to health after a non-malicious failure or deliberate compromise, even in cases where the precise cause of the failure or compromise are not understood. A recovery sequence might begin by restoring only a few components to a healthy state, and move on my restoring more and more components until the overall application can once again operate normally. In the worst case in which all components must be reset, Microreboot is essentially equivalent to traditional reboot. However, in cases where Microreboot overcomes a failure by resetting only some of an application's components, Microreboot preserves the state of the balance of the components—state which would have been needlessly discarded in a traditional reboot.

Candea formulated his notion of Microreboot in the context of "Crash-only" Java applications [3]—that is software whose architecture meets the following peculiar

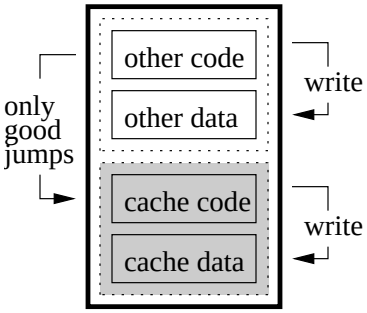

Figure 1: Rio File Cache fault domains

requirements:

1. Applications must be composed of many small co-processes.
2. Co-processes must communicate only through well-defined message-passing interfaces.
3. All request messages must be retryable.
4. All leases must be expirable.
5. All critical state must be stored in an external transactional database.

Given an application whose architecture meets these Crash-only requirements, one can apply Microreboot by resetting individual co-processes to their initial state. The above Crash-only requirements are designed to ensure that co-processes can survive the temporary loss of their peers and can resume cooperating with peers that have been reset. For example, since all leases are expirable, individual co-processes that fail to release locks due to resets will not permanently upset the system, since their lease will expire and the lock will be taken from them automatically.

It would be useful to apply Microreboot to legacy C applications, if a means of retrofitting an existing application to meet Crash-only requirements could be found. This report presents the results of an experiment to use SFI techniques in the manner of the Rio File Cache to provide guarantees equivalent to the above external transactional database requirement without actually relying on an external database. The Rio File Cache and its use of SFI is described in the next section.

3 The Rio File Cache

Chen and others [5] demonstrated the Rio File Cache in a modified Digital UNIX kernel on the Digital Alpha platform. In a traditional Digital UNIX system, the contents

of the operating system's buffer cache can be arbitrarily modified (and consequently rendered useless) by a kernel that has experienced a failure and misbehaves in an unexpected way while crashing. The Rio File Cache uses Software Fault Isolation [22] (SFI) to ensure that kernel code outside of the buffer cache's proper access functions cannot modify the contents of the buffer cache, thereby preserving the buffer cache's contents during a crash. Figure 1 contains a highly-simplified diagram of the modified Digital UNIX kernel. The modified kernel is split into two SFI "fault domains": one containing the buffer cache, its metadata, and its access functions (shaded) and the other containing the remainder of the kernel's code and data (unshaded). The authors used SFI and static analysis to argue that only cache code can write to cache data, and the non-cache code can call cache accessor functions only in cases deemed proper by the authors.

Because the Digital UNIX buffer cache is never paged and always resides in the same region of physical RAM, it is possible for a new instance of the kernel to re-use the contents of the buffer cache left by the previous instance after reboot, provided that the contents were not ruined during a crash, and that there is some means of making the kernel's state stored outside of the buffer cache consistent with the buffer cache's contents. The authors use SFI to ensure the integrity of the buffer cache itself, and store additional metadata along with the buffer cache to enable the new kernel instance to "synchronize" its state with the buffer cache's contents.

As described in section 2, this report describes the results of an effort to use SFI in the manner of the Rio File Cache to preserve critical state that in a Crash-only application would be stored in an external database. The experimental SFI implementation is described in the next section.

4 Experimental implementation

This section describes the implementation of a experimental C application that uses a simple but non-trivial data structure, and the application of Rio File Cache-style SFI to it in order to preserve both the data structure's data and metadata across software restart. Software restart refers to returning a program to some prior healthy state without actually terminating its process, perhaps by a combinator of `setjmp` and `longjmp` calls in C. The primary goal of this implementation is to provide a simple kind of existence proof—one simple example of a C application that uses Rio File Cache-style SFI to preserve its critical state in the same way the use of a transactional database does in Crash-only software (Crash-only requirement 5 from section 2).

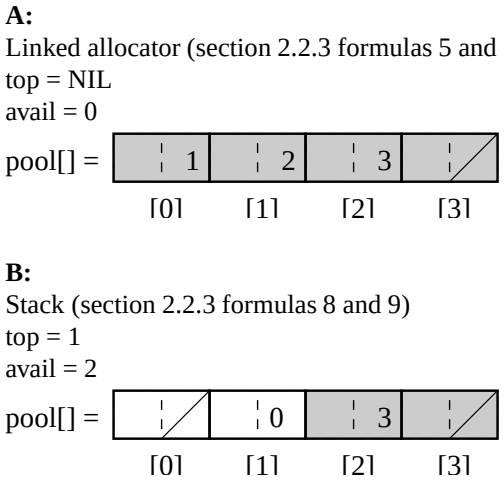

Figure 2: Knuth linked allocator and stack

4.1 The experimental application

The experimental application implements the linked stack algorithm from Knuth [15], chosen as the first algorithm in Knuth that required the preservation of data at two levels of abstraction (in order to parallel the preservation of both data and metadata in the Rio File Cache.) This algorithm includes two parts: first, a linked allocator (Knuth section 2.2.3 formulas 5 and 6) and second, an application stack that uses this linked allocator for dynamic memory allocation (Knuth section 2.2.3 formulas 8 and 9). These data structures are diagrammed in figure 2.

The linked allocator is shown in figure 2A. The linked allocator is a heap management scheme similar to the C library's `malloc` function, with the exception that allocated regions of memory are always the same size. To implement this `malloc`-like functionality, the linked allocator uses a large array of constant-sized cells, labeled `pool[]` in the diagram. Each cell has two parts divided by a dashed vertical line in the diagram. The left part of each cell holds application data. The right part contains either the index of another node in the array or the distinguished value NIL (called Delta in Knuth). At program start, all pool cells are linked into a singly-linked list called the "available list". A variable avail holds the index of the first cell in the available list. The linked allocator has two accessor functions: `allocate` and `delete` which allocated cells for application use by removing them from the available list and return them to the available list for later reuse, respectively.

The stack is shown in the diagram in figure 2B. The stack is implemented as a linked list that is similar to the available list. The index of the cell at the top of the stack is stored in the variable `top`, each cell contains the index of the cell beneath it in the stack, or in the case of the

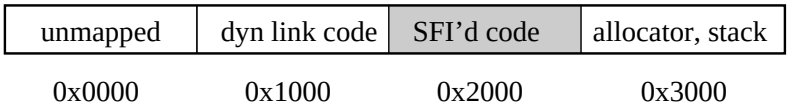

Figure 3: Experimental fault domains

cell at the bottom of the stack, NIL. The stack has the two traditional accessor functions, `push` and `pop` that use the linked allocator's `allocate` and `delete` functions to manage memory. The diagram shows a stack with two cells allocated, 1 on top and 0 on bottom. Two pool cells, 2 and 3, remain unallocated.

4.2 Fault domains

Figure 3 contains a diagram of the fault domains chosen for SFI sandboxing in the experimental application. The experimental application is a demand-paged MAC OS X 10.3 application and is three 4KB pages long. The diagram shows the three pages plus an additional "zero" page along with the lower two bytes of their addresses. Page `0x0000` is left unmapped by the linker/loader so that dereferences of NUL pointers will, upon reaching this unmapped page, fault. Page `01000` contains dynamic linker code added by the compiler. Page `0x3000` contains the linked allocator and stack portion of the application—the portion to be protected from accidental modification during Microreboot. Page `0x2000` (shaded in the diagram) contains the remainder of the experimental application code—the code which must be controlled with SFI. The SFI implementation uses the 4th byte of the page address (0,1,2,3) to identify fault domains. To maintain consistency with the original Wahbe SFI terminology, the following description of the actual SFI implementation refers to these numbers as "segment numbers" and fault domains as "segments". Segment 2 is the segment to which SFI will be applied, as described in the next subsection.

4.3 SFI implementation

This subsection describes the application of Wahbe's "sandboxing" SFI to segment 2 of the experimental application. Sandboxing SFI required the addition of three kinds of PowerPC assembly code to the existing application code: the initialization of reserved registers at the beginning of the main program, the sandboxing of branch (jump) statements to addresses stored in registers when returning from functions, and the sandboxing of stores to addresses stored in registers. All assembly code was added manually to the C source files using the inline assembly support provided by the Mac OS X version of the GNU C compiler. Each of the three cases are described below.

SFI initialization Table 1 shows the assembly code added to the beginning of the experimental application's main function to set the reserved mask register (`r17` on line 11) and the reserved segment register (`r18`) on line 12). The `0x0fff` mask is set to mask out the segment number residing in the 4th byte of the addresses shown in figure 2. The segment is set to `0x2000`—the only proper segment number for branch and store instructions residing in fault domain 2.

These registers are "reserved" only in the sense that only the sandboxing SFI uses them, as required by the Wahbe technique. However, the compiler was not instructed to avoid using them in application code. Instead, the application used so few of the architecture's 32 general-purpose registers that it was a simple matter to examine a disassembly of the experimental application and choose registers for the SFI code that the compiler had simply not chosen to use for the application code.

Branch sandboxing Table 2 shows the assembly code added to sandbox branch instructions at the end of functions in segment 2 code. The goal of this sandboxing is to ensure that no code in segment 2 can branch to another fault domain. The assembly code accomplishes this prevention by taking the return address from the special register `lr` and placing it in `r16`, the register reserved for sandbox calculations (line 22). The table code's comments describe an example where the return address in `r16` is `0x00002294`—an address that is legal but will nevertheless be sandboxed. The assembly code proceeds to mask out the segment number on line 23, and replace it with the segment number for segment 2 on line 24—the same value that was there before, since the example describes a well-behaved branch.

At this point the assembly code must deviate from the Wahbe technique by returning the sandboxed address back to the special `lr` "link" register (line 25) and then branches to the address in `lr` (line 26). According to the Wahbe technique, the assembly code should have branched to the address contained in `r16`, allowing one to make the argument that all sandboxed branches use `r16`—a register used only by SFI code. However, although the Alpha instruction set used by Wahbe apparently contained an instruction causing a branch through

```
10          /* from main: dedicated sandboxing registers */
11    li         r17, 0x0fff      /* r17 is mask register */
12    li         r18, 0x2000   /* r18 is segment register */
```

Table 1: Initializing sandbox registers

```
20           /* perform sandboxing SFI on return branch */
21           /* r16, wahbe dedicated reg =  return addr */
22    mfspr      r16, lr            /* r16 = 0x00002294  */
23    and        r16, r16, r17      /* r16 = 0x00000294  */
24    or         r16, r16, r18      /* r16 = 0x00002294  */
25    mtspr      lr, r16            /* lr = r16          */
26    blr                         /* ‘‘branch to lr’’    */
```

Table 2: Sandboxing function return branch

```
30    li         r15, 0x42           /* put ’B’ into r15 */
31
32               /* perform wahbe sandboxing SFI on store */
33               /* r16, wahbe dedicated reg = store addr */
34                     /* r0  = store addr = 0x0000302b  */
35    and        r16, r0, r17       /* r16 = 0x0000002b  */
36    or         r16, r16, r18      /* r16 = 0x0000202b  */
37    stb        r15,0x0(r16)       /* store to FD2 text */
```

Table 3: Sandboxing stores through registers

```
01    Pool index:  0    1    2    3    4    5    6    7
02    Available ->                    Z -> Z -> Z -> Z
03    orphans:
04                  A <- A <- A <- A                    <- Stack
05             a) push data onto stack
06             b) pop data off of stack
07             c) munge stack, no SFI
08             d) munge stack, with SFI
09             e) reboot with good integrity
10             f) reboot with half push (stack link)
11             g) reboot with half push (avail link)
12             h) reboot with half pop (stack link)
13             i) reboot with half pop (avail link)
```

Table 4: Demo program output with 4 nodes allocated on stack

a general-purpose register such as `r16`, the PowerPC instruction set does not have such an instruction. Instead, it has only the `blr` instruction to branch to the address in the special-purpose `lr` register used on line 26. This necessity complicates the sandboxing argument—one must argue that although the application code also uses the `lr` register, the SFI code always sets it properly—the sandboxing is still effective.

Store sandboxing The assembly code in table 3 shows the instructions added to sandbox stores. The assembly code's comments show an illegal attempt to modify a byte in segment 3—the application stack's segment that must be protected. Specifically, it attempts to write a `'B'` to byte `0x0000302b`, as shown on line 30. As in the above branch case, the original store address is loaded into `r16` and the the original 4th byte that indicates the segment number is masked out (line 34). Line 36 replaces the original 3 with a 2, causing the store byte instruction on line 37 to modify location `0x0000202b` in segment 2 rather than `0x0000302b` in segment 3. The location in segment 2 happens to be read-only program text, and the sandboxed store causes the OS to trap a memory access error and send a SIGBUS signal to the experimental application. The experimental application catches and handles the signal, and it's application stack data and linked allocator data remain unharmed.

5 Microreboot and consistency

This section describes the experimental application's Microreboot behavior, the state consistency problems it raises, and solution strategies for these consistency problems.

5.1 Microreboot control flow

Figure 4 contains a diagram of the experimental application's control flow. It begins with the square marked `init`, where all of the its data structures are initialized. This is where the linked allocator's "available" list is initialized to include all of the cells in the pool, and the application stack is set to be empty. Once initialization is complete, the experimental application moves to the shaded square marked `good`. This square represents the healthy state to which Microreboot returns the experimental application after a failure. Because the earlier initialization is not repeated, the linked allocator and application stack state survives Microreboot.

After the reaching the `good` state, the experimental application moves on the the `user input` square, where it loops indefinitely handling user `push` and `pop` requests. There is also the `SIGBUS` option which tests the experimental application's SFI sandboxing functionality that, as described in the previous section, results in the receipt of a `SIGBUS` signal. In addition, there is an explicit `boot` option to cause a Microreboot. Both the `SIGBUS` and `boot` squares in the control flow diagram lead back via dashed lines to the `good` square—these dashed lines trace the control flow of Microreboot.

Figure 4 shows the screen output of the experimental application. The experimental application's linked allocator pool has eight cells. Output lines 1–4 form a diagram of the pool's state at the time the output was produced. Line 1 shows the pool cell indices. Line 2 shows the linked allocator's available list containing cells 4 through 7, with the data portion of each cell set to `'Z'`. Line 4 shows the remaining cells allocated for the stack, with cell 3 being the top. Each stack cell's data portion is set to `'A'`. The remaining lines 4–13 list the various interactive commands available to the user.

5.2 Consistency issues

The experimental application uses sandboxing SFI to provide the isolation required to enable its application stack and linked allocator data to persist unharmed across Microreboots. It does so to provide isolation and persistence equivalent to that provided by the use of external transaction-oriented databases in Crash-only software (requirement 5 in section 2). While section 4 describes its effectiveness in this regard, isolation is not the only feature provided by the external transaction-oriented databases.

The other critical feature is consistency—the external transaction-oriented databases ensure that if a Microreboot occurs while the application is in the middle of updating its critical data structures, the data structures will be left in a consistent state. Although the experimental application's use of sandboxing SFI provides isolation, it does not provide such consistency guarantees.

To illustrate this lack of a consistency guarantee, figure 5 contains a sequence of diagrams of the experimental application's linked allocator pool and application stack. The diagrams show the state changes that occur during a `push` operation.

Diagram 0 shows the state of the pool and application stack before the `push` operation begins. This state is consistent: the stack contains cells 1 and 0, and the linked allocator's available list contains cells 2 and 3. Because the pool and application stack are in a consistent state, a Microreboot at this point will cause no consistency problems.

Diagram 1 shows the state that results just after the beginning of the `push`. The `push` function has called

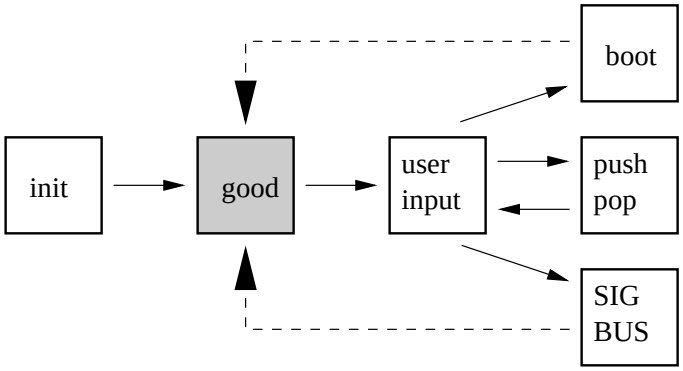

Figure 4: Experimental program control flow diagram

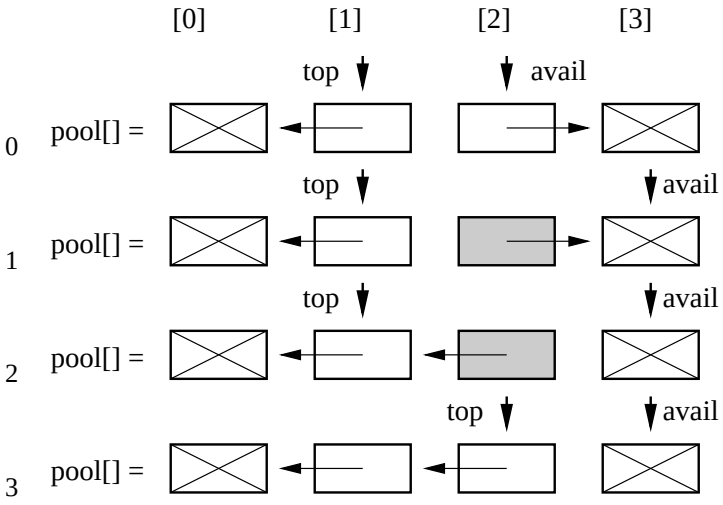

Figure 5: Intermediate pool and stack states during push operation

the linked allocator's `allocate` function to allocate a new cell for the stack. This new cell is shaded in the diagram. The `allocate` function has removed the new cell from its available list, but the `push` function has not yet begun to initialize the new cell. If a Microreboot occurs with the pool and application stack in this state, the experimental application will recover to find its pool and application stack in an inconsistent state. As the diagram shows, the new (shaded) cell is "orphaned"—it is neither in the stack's list or the linked allocator's available list.

Diagram 2 shows the state that results near the end of the `push`. The `push` function has set the new (shaded) cell's link to point to the old top of the stack, but has not yet set its top pointer to the new cell. As in the previous diagram, a Microreboot at this point will also result in an orphaned new cell.

Diagram 3 shows the consistent state that results when the `push` is complete. A Microreboot at this point will cause no consistency problems.

Figure 5's diagram shows two states that result in a pool cell being orphaned on Microreboot. Each orphaned cell is neither allocated nor free. Over the course of many Microreboots, the entire available pool may be orphaned, and no more `push` operations will be possible.

There are a variety of strategies for dealing with consistency problems after Microreboot in legacy C applications. Some are described below:

transactions: One might add code to the application to make all accesses to critical data transactional. Although this addition might be accomplished using interposition [12, 10] of some kind, it is not hard to imagine cases where this solution would be a complex, invasive, and difficult process.

just wait: One might simply cause Microreboots only at points in the application's control flow where its critical data structures are in a consistent state (such as diagrams 0 and 3 in figure 5. Although this strategy has the benefit of simplicity, there may be no guarantee that a given application will ever reach such a consistent state after a failure that demands Microreboot.

fsck: One might apply a filesystem check "fsck-like" consistency check and repair on Microreboot. Less invasive than the transaction strategy, without the unbounded waiting concerns of the waiting strategy, this is the strategy implemented in the experimental application.

Note that all of these strategies require some kind of analysis and understanding of the application's code to determine exactly what consistent state is. This question bears on the secondary results of this work as described in section 8.

Table 5 shows the output of the experimental application as a Microreboot is manually initiated while the application stack is in mid-`push`—specifically, in the state described by diagram 2 in figure 5. Lines 20–23 shows the linked allocator pool and application stack in a consistent state. Line 33 shows the user input that initiates the Microreboot. Lines 34–38 shows the state of the linked allocator pool and application stack on Microreboot. Note that cell 4 is orphaned—it is on neither the stack or available lists.

Lines 39–45 show the operation of the experimental application's fsck-like integrity checking and repair function. This function finds orphaned nodes and returns them to the available list. Just as a transactional database ensures that the results of a transaction will either be completely committed or not committed at all, the fsck-like routine ensures that the results of a push or pop will either be entirely complete or not apparent at all.

The experimental application's fsck-like routine's algorithm was constructed by a human analyst after a manual examination of the experimental application's `allocate`, `delete`, `push`, and `pop` functions. The method used was as follows. First, the analyst constructed the code shown in table 6 by taking the application stack functions and inlining the linked allocator functions. This combined version of the experimental application's code made it easier for the analyst to visualize the instruction stream as it occurs sequentially over time. In the table, the linked allocator's inlined statements are double-indented on lines 65–66 and 89–91. The remaining statements are application stack code.

The analyst then performed an informal abstract execution of the `push` and `pop` functions using pencil and paper, methodically imagining the states that would result if execution was stopped by a Microreboot between each line. The result was a set of diagrams similar to the simplified examples shown in figure 5. The analyst `ORPHAN_AVAIL` and `ORPHAN_TOP` macros at strategic points in the code. These macros allow the user to cause Microreboots at these points in the control flow, causing specific kinds of inconsistencies to test the fsck-like function, as shown in the output contained in table 5.

Having constructed a list of all the possible inconsistent linked allocator and application stack states, the analyst then relied on intuition to determine that the key to *detecting* consistency problems was to find orphaned nodes—that is, node that were neither in the linked allocator's available list nor the application stack's list. The analyst then noted that it was impossible to distinguish between a cell orphaned by an incomplete `push` and one orphaned by an incomplete `pop`. Both came in two fla-

```
Pool index:  0    1    2    3    4    5    6    7
Available ->                     Z -> Z -> Z -> Z
orphans:
             A <- A <- A <- A                    <- Stack
        a) push data onto stack
        b) pop data off of stack
        c) munge stack, no SFI
        d) munge stack, with SFI
        e) reboot with good integrity
        f) reboot with half push (stack link)
        g) reboot with half push (avail link)
        h) reboot with half pop (stack link)
        i) reboot with half pop (avail link)
f
State on microreboot:
Pool index:  0    1    2    3    4    5    6    7
Available ->                          Z -> Z -> Z
orphans:                         A:3
             A <- A <- A <- A                    <- Stack
Integrity check:
        orphaned node 4... returned to available pool.

Pool index:  0    1    2    3    4    5    6    7
Available ->                     A -> Z -> Z -> Z
orphans:
             A <- A <- A <- A                    <- Stack
        a) push data onto stack
        b) pop data off of stack
        c) munge stack, no SFI
        d) munge stack, with SFI
        e) reboot with good integrity
        f) reboot with half push (stack link)
        g) reboot with half push (avail link)
        h) reboot with half pop (stack link)
        i) reboot with half pop (avail link)
```

Table 5: Demo program output after consistency check and repair

```
60    void
61    stack_push(info_t y) {
62
63           index_t p;  /* temp pointer to new stack node */
64
65                   p = avail;
66                   avail = LINK(avail);
67
68            INFO(p) = y;
69            ORPHAN_AVAIL;
70            LINK(p) = top;
71            ORPHAN_TOP;
72            top = p;
73
74    } /* stack_push() */
75
76
77
78    info_t
79    stack_pop(void) {
80
81            index_t p;  /* temp index of top stack node */
82            info_t  y;  /* info (data) to return to caller */
83
84            p = top;
85            top = LINK(p);
86            y = INFO(p);
87            ORPHAN_TOP y;
88
89                   LINK(p) = avail;
90                   ORPHAN_AVAIL;
91                   avail = p;
92
93            return y;
94
95    } /* stack_pop() */
```

Table 6: Partial stack push and pop code with pool code inlined

vors: one with a link to the available pool, and one with a link to the stack.

The analyst relied on his understanding of the application stack's semantics to determine that the key to *repairing* consistency problems was to return orphaned cells to the available pool—essentially completing an incomplete `pop`, regardless of whether the interrupted operation was a `push` or `pop`. Mistaking an incomplete `push` for a `pop` does lose the `push` data, but it also returns the stack to the consistent state before the `push` began. Mistaking an incomplete `pop` as a `push`, by comparison, would put the stack into a state that was never intended by the application.

It would be desirable to automate the task of generating fsck consistency problem detection and repair, if possible. This issue is discussed in section 8.

6 Related Work

Although the primary focus of this project was to demonstrate the use of SFI to meet some of the Crash-only software requirements required to enable Microreboot, the consistency issues that arose arguably became the more interesting aspect of the work. These issues are described in section 5. This section briefly summarizes some work related to the task of identifying and repairing consistency (integrity) problems in program state during runtime.

6.1 Recovery by restart

Dunlap and others have demonstrated the use of virtual machine monitors to recover compromised systems by "rolling programs back" to a previous healthy state [9]. In their "ReVirt" system, the virtual machine monitor produces extensive logs of virtual host activity. It uses these logs to analyze intrusions. It can also use these logs to perform repair on individual processes by rolling back sequences of events as a database might. ReVirt returns entire programs to a prior healthy state that actually existed at some point in history. Because the program had reached that state before, one can argue that the state is necessarily consistent.

In contrast, Microreboot manufactures an artificial healthy state in which some of a program's state is restored to a prior state and some is not. The consistency problems in section 5, for example, occur because the stack may be left in a mid-push state but the program counter will be reset rather than restored to the corresponding midpoint of the push function. Consequently, Microreboot can preserve more state, but must deal with more consistency issues than ReVirt. ReVirt is directly applicable to systems running legacy C programs, while this report demonstrates how to meet only some of the Crash-only requirements needed for Microreboot of legacy C programs.

6.2 Consistency specification

Demsky and Rinard have developed a specification language for data structure definition and runtime constraints, along with a program generator that outputs programs containing extra code to automatically detect and repair their own data structures when they fail to meet the specified constraints [7]. This work places one level of abstraction on top of well-known historical work with the 5ESS [14] and MVS [16] systems: in those systems, the inconsistency detection and repair procedures were coded manually. In this work, they are generated automatically from manually-written specifications. In later work, Demsky and Rinard explore static program analysis techniques to determine whether or not a given set of consistency constraints will result in a repair procedure that will always terminate [8].

Nentwich and others have also investigated techniques similar to Demsky and Rinard [18]. They have developed xlinkit, a tool that detects inconsistencies between distributed versions of collaboratively-developed documents structured in XML [2]. It does so based on consistency constraints written manually in a specification language based on first order logic and XPath [6] expressions. These constraints deal with XML tags and values, such as "every item in this container should have a unique name value." In later work [19], they describe a tool which analyzes these constraints and generates a set of repair actions. Actions for the above example might include deleting or renaming items with non-unique names. Human intervention is required to prune dumb repair actions from the list, and to pick the most appropriate action from the list at repair time.

As explained in section 5, the consistency constraints described in this report were derived manually (as in the above related work) but were not specified formally except through the entirely-manual coding of the fsck-like routine (in contrast to the above use of specification languages and program generators). In the future, it would be helpful to have tools that automate at least part of the constraint-derivation process, as described in section 8.

6.3 Runtime integrity monitors

Many authors have described external monitors that examine the state of running programs for consistency problems, including Livewire [11], Backdoors [21, 1], Copilot [20], and the work of Grizzard and others on "whitehat rootkits" [13]. Some of these monitors are implemented as kernel processes, others in virtual machine

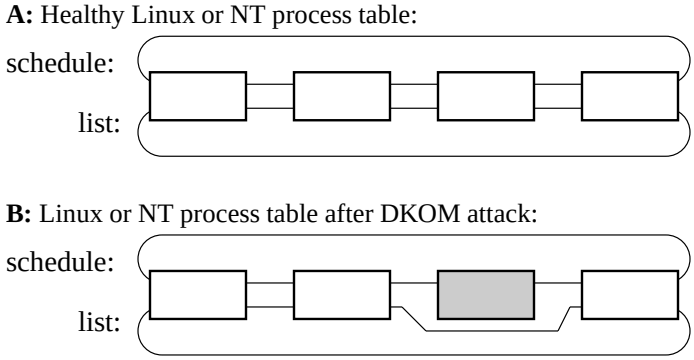

Figure 6: Linux or NT process table before and after DKOM attack

monitors, and still others on PCI add-on cards intended to be installed in the system to be monitored.

As a rule, these monitors are capable of monitoring and perhaps even repairing the consistency of data structures that are involved in known popular attack vectors, but their coverage does not extend to all data structures of interest. Tools to help automate the process of deriving consistency constraints, such as those imagined in section 8, would be of great help in expanding the coverage (and consequently the effectiveness) of these runtime integrity monitors.

7 Conclusions

The primary result of this work is a kind of simple existence proof: The experimental application represents the successful application of SFI [22] techniques to isolate a critical data structure intended for persistence across Microreboot in a fashion similar to the Rio File Cache [5]. The goal of this exercise was to show that Rio File Cache-style SFI might be used to give legacy C applications functionality equivalent to the use of external transaction-oriented databases mandated for Crash-only software [3] (Crash-only requirement 5 from section 2). As described in section 5, this meeting this goal required more than just SFI-provided isolation. It also required additional consistency-preserving functionality to match the integrity-preserving transactional features of the database.

Although the experimental application is quite simple, it does involve the preservation of both the state of an application data structures (the stack) and its metadata (the linked allocator state). This coverage provides some hope that the technique might be applicable to larger C applications, as well.

There are also a number of secondary results: First, the use of SFI might also enabled a legacy C application to meet the Crash-only requirement that cross-domain calls be made through strong interfaces (requirement 2 in section 2)—that is, that jumps can't be made from one domain to an arbitrary instruction in another. The use full use of SFI demands that a static analysis be performed on the application to determine which cross-domain branches (calls) are legal. The use of sandboxing described in section 4 merely handles the cases that would be difficult to analyze statically— the static analysis is what provides the most meaningful guarantees. Presumably, this static analysis would ensure that a legacy C application provided a guarantee equivalent to the Crash-only mandate for strong interfaces. However, this argument is largely conjecture as no complete static analysis of the experimental application was attempted.

Second, as described in section 4, the lack of a "branch to an address stored in a general-purpose register" instruction in the PowerPC architecture requires some adjustment to the SFI technique described by Wahbe and others. Although the static analysis is complicated somewhat, the technique remains effective.

Finally, it must be noted that although the success of the experimental application shows that progress is possible, it by no means should be taken as an indication that applying this technique to a large poorly-documented legacy C application would be in any way easy or easily-automated. The static analysis might be particularly difficult, as it requires first deciding how to partition the application into fault domains, and then deciding which cross-domain calls are legal and which are not. Both of these tasks might require considerable understanding of how the application works; such understanding may be difficult to come by, particularly where "legacy" applications are concerned.

8 Future Work

Section 6 briefly introduced the notion of a runtime integrity monitor—a program or device that examines the state of a running application or operating system ker-

nel for consistency problems and, in some cases, repairs them. Arguably, these devices work best when monitoring data structures that should rarely or never change during runtime. Examples of these largely-static data structures are numerous: in kernels they include jump tables such as system call vectors and interrupt descriptor tables—both historically popular targets for malicious tampering.

However, recent more-sophisticated attacks have begun to focus their tampering efforts on more-dynamic data structures which are arguably harder for runtime execution monitors to effectively examine and repair. Perhaps the best example of these "Direct Kernel Object Manipulation" (DKOM) attacks concerns the process table. Figure 6A shows a conceptual diagram of the process table of a Linux or Windows (NT) kernel. As shown in the diagram, the table actually takes the form of a set of nodes tied together in two circular linked lists. One circular list is traversed by the scheduler while choosing processes to run. The other circular list is traversed by the process-listing code which produces output for the GNU/Linux ps command or the Windows Task List.

Figure 6B shows one common tampering strategy used by attackers who wish to hide a process from legitimate administrators is to remove their process (shown shaded) from the process-listing circular linked list while leaving it in the scheduler's circular linked list. Because the process remains in the scheduler's circular linked list, the scheduler still allows it to run. But because it is no longer in the process-listing circular linked list, it does not appear in process listing output visible to legitimate administrators.

After some manual examination of the diagrams in figure 6, one might reasonably and correctly conclude that "each process's entry must be included in both circular linked lists" is a useful consistency constraint in this situation. However, one must note that a run-time integrity monitor may see only snapshots of the state of the process table during run-time. Because the state of the process table is changing continuously, such snapshots may portray states that violate this consistency constraint for no reason other than the kernel happened to be adding or removing a process at the moment the snapshot was taken.

The fact that harmless but inconsistent states may be visible to an external monitor raises some of the same consistency issues discussed in section 5. Just as it would be helpful to have a tool which helps to determine both the consistent and all of the intermediate inconsistent states of a given data structure for the sake of fsck-writing, it would also be helpful to have such a tool for the sake of writing monitoring code that can make sense of arbitrary snapshots.

Section 6's discussion of related work noted multiple explorations of systems where human analyst manually derive consistency constraints and then use automated tools to generate fsck-like code from them. It would also be helpful to have tools that could automatically derive consistency constraints, perhaps after being given some hints by a human analyst. Failing that, it might also be helpful to have tools which could confirm or deny an intuitive guess at a constraint made by a human analyst working in haste from an imperfect understanding of the application in question. Although such tools may ultimately be unobtainable, these topics seem deserving of future exploration.

This discussion leaves a number of open questions: Does the manual creation of the fsck routine for the experimental application described in subsection 5.2 provide any insight into the problem of expressing consistency constraints for constantly-changing data structures? Is the fsck routine itself akin to a formal description of such a constraint? Are all in-memory data structures simple filesystems in need of an fsck routine? Subsection 5.2's manual constraint-discovery method involved imagining the insertion of Microreboots between each step in the critical data structure's access routines. This approach seems similar to the more formal method of Myers and others [17] to reason about their notion of Robust Declassification, where they modeled the ability of an attacker to change a running program's state by allowing the insertion of arbitrary new program statements between the existing ones. Can something like the Myers method be used to formalize reasoning about consistency issues?

References

[1] Aniruddha Bohra, Iulian Neamtiu, Pascal Gallard, Florin Sultan, and Liviu Iftode. Remote Repair of Operating System State Using Backdoors. In *Proceedings of The International Conference on Autonomic Computing (ICAC-04)*, May 2004.

[2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language. Recommendation REC-xml-20001006, World Wide Web Consortium, October 2000.

[3] G. Candea and A. Fox. Crash-only software. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Operating Systems*, May 2003.

[4] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot - A Technique for Cheap Recovery. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, December 2004.

[5] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 7th*

ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), October 1996.

[6] J. Clark and S. Derose. XML Path Language (XPath) Version 1.0. Recommendation REC-xpath-19991116, World Wide Web Consortium, November 1999.

[7] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors and Data Structures. *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 38(11), October 2003.

[8] Brian Demsky and Martin Rinard. Static Specification Analysis for Termination of Specification-Based Data Structure Repair. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, November 2003.

[9] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[10] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 2–16, Berkeley, California, May 1999.

[11] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Annual ISOC Network and Distributed System Security Symposium*, February 2003.

[12] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 1998.

[13] Julian B. Grizzard, John G. Levine, and Henry L. Owen. Re-establishing Trust in Compromised Systems: Recovering from Rootkits That Trojan the System Call Table. In *Proceedings of European Symposium on Research in Computer Security (ESORICS)*, pages 369–384, Sophia Antipolis, France, September 2004.

[14] G. Haugk, F. Lax, R. Royer, , and J. Williams. The 5ESS switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385 – 1416, July-August 1985.

[15] Donald E. Knuth. *The Art of Computer Programming—Volume 1/Fundamental Algorithms*. Addison-Wesley, 1968.

[16] Samiha Mourad and Dorothy Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, 13(10):1135 – 1139, October 1987.

[17] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing Robust Declassification. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004.

[18] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151 – 185, May 2002.

[19] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proceedings fo the 25th International Conference on Software Engineering*, May 2003.

[20] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot – a Coprocessor-based Kernel Run-time Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[21] F. Sultan, A. Bohra, P. Gallard, I. Neamtiu, S. Smaldone, Y. Pan, and L. Iftode. Nonintrusive Remote Healing Using Backdoors. In *Proceedings of the 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[22] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. *ACM SIGOPS Operating Systems Review*, 27(5), December 1993.