# Handling Self-Modifying Code Using Software Dynamic Translation

Joy W. Kamunyori
University of Virginia
jkamunyori@cs.virginia.edu

## 1. Problem and Motivation

The term self-modifying code refers to code that changes or updates its own instructions during its execution. Self-modifying code is widely used in runtime code generation in Just-In-Time (JIT) compilers [1,2]. Self-modifying code is also seen in applications such as Adobe Premiere, and in games like Doom [2]. The ability to self-modify is therefore an important and useful programming technique. Self-modifying code can also be used for malicious purposes, such as in a virus that avoids detection by changing itself in every generation, or by malicious software that attempts to modify a running program for detrimental purposes [8].

Strata, our software dynamic translator (SDT), was not originally written to support self-modifying code, a significant limitation for any software dynamic translator, as it rules out an entire class of applications from dynamic translation. Extending a software dynamic translator such as Strata to handle self-modifying code safely and reliably is a complex procedure. It is complex because once an instruction is fetched and saved in the fragment cache, subsequent occurrences of the instruction result in the cached copy of the instruction being executed. Without a way to notify Strata when an instruction has been modified, the modified instruction will not be retranslated, so the outdated cached copy will continue to be executed. This work focuses on our extension of Strata to recognize and appropriately handle self-modifying code, which involves developing a notification procedure to alert the SDT when an instruction is modified. This notification procedure is implemented using memory protection and signal delivery mechanisms which act as an impetus for Strata to flush the fragment cache and start retranslating instructions from the point of modification.

This paper discusses the process of extending Strata to support self-modifying code in client applications, and is organized as follows. Section 2 provides a brief overview of the Strata software dynamic translation infrastructure, and presents a simple test example of self-modifying code. Section 3 describes an overview of the approach developed in order to correctly handle this programming technique, while Section 4 evaluates our approach and concludes.

## 2. Background and Related Work

### Strata Overview

Strata is a portable, extensible software dynamic translation (SDT) infrastructure, developed at the University of Virginia. It can be used for several purposes, such as binary translation, program profiling and improving application security. Strata works by loading a binary application dynamically, and mediating its execution on a machine. This mediation is accomplished by examining the application's instructions and translating them before they are executed on the host CPU. The translated instructions are stored in a cache managed by Strata, called the fragment cache, where instructions are organized in fragments, which are sequences of code in which branches can appear only at the end [3,4,5,7].

Figure 1 is a high-level representation of how the Strata infrastructure works. Before entering the Strata Virtual Machine, the application context (e.g. PC, condition code, registers, etc.) is captured and saved. Once the context has been captured, the next application instruction is processed. If a translation for this instruction already exists in the fragment cache, the application context is restored, and Strata begins executing the cached translated instructions on the host CPU. If there is no translated instruction for the application instruction in the fragment cache, Strata allocates space for a new fragment in the cache, and begins to populate the fragment by fetching, decoding and translating application instructions until an end-of-fragment condition is met. This end-of-fragment condition usually involves encountering a branch instruction in the application. Once the end-of-fragment condition is met, the application context is restored, and the newly translated fragment is executed [3,4,5,7].

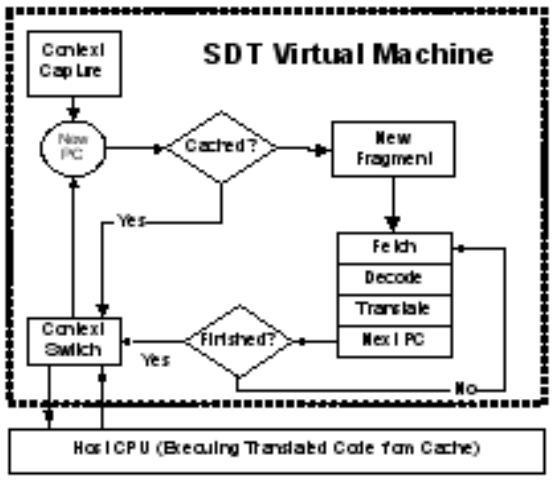


**Figure 1: The Strata Framework**

### Test Case

We developed a test case to illustrate that Strata did not handle self-modifying code, and to drive discovery of what modifications were necessary to fix this shortcoming. The test case is a simple example of self-modifying code. Figure 2 contains the test case code. The test case is a simple program that makes a direct system call to write to standard out by making an indirect call to a data buffer. Lines 8-12 in the figure show the array *testcpy*, which contains the x86 binary executable code of the assembly program in lines 1-6. This program loads the write system call number, and the file descriptor for standard out (lines 1-2), and then prints the string located at the address stored in register *ecx* (line 3) when the system call is made (line 5). Originally, the address loaded into *ecx* is that of the string *test* on line 14.

At lines 21 and 22, a buffer called *virus* is created and aligned to the beginning of a page. The buffer *virus* is loaded with the contents of the *testcpy* array at line 25, and then the buffer is made executable (lines 27-28). At line 31, *virus* is executed, so that the words "Hello world" are output to the screen. The address and length of the string to be printed is then changed in the buffer (lines 34-35), to that of the string *virusstr* at line 15, and *virus* is executed again (line 37) so that the string "HELLO VIRUS!!" is output.

```
/* assembly code of testcpy

1. mov $0x4,%eax ;load write system call
2. mov $0x1,%ebx ;load file descriptor for standard out
3. mov $0x80d1583,%ecx ;load address of string
4. mov $0xc,%edx ;load length of string
5. int $0x80 ;make system call
6. ret
7. */
8. unsigned char testcpy[] = { 0xb8, 0x04, 0x00, 0x00, 0x00,
9.  0xbb, 0x01, 0x00, 0x00, 0x00,
10.  0xb9, 0x63, 0x15, 0x0d, 0x08, /* The address of test is 0x080d1563
     */
11.  0xba, 0x0c, 0x00, 0x00, 0x00,
12.  0xcd, 0x80, 0xc3 };
13.
14. char test[] = "Hello world\n";
15. char virusstr[] = "HELLO VIRUS!!\n";
16.
17. unsigned char *virus;
18.
19. int main(){
20.  int i;
21.  virus = malloc(1024 + PAGESIZE-1);
22.  virus = (char *)((int) virus + PAGESIZE-1) & ~(PAGESIZE-1));
23.
24. /* loading program code to buffer */
25.  memcpy(virus, testcpy, 35);
26.
27. /* making the buffer executable */
28.  if(mprotect(virus, 1024, PROT_READ|PROT_EXEC|PROT_WRITE))
```

```
29. printf("Couldn't mprotect\n");
30.
31. ((void (*)())virus)();
32.
33. /* changing the address and length of the string */
34. virus[11] = 0x70; /* address of virusstr is 0x080d1570 */
35. virus[16] = 0x0e; /* the string is 14 bytes */
36.
37. ((void (*)())virus)();
38.
39. }
```

**Figure 2: Self-Modifying Code Test Case**

This test case proved to be a challenge to Strata due to the fragment-caching mechanism. Fragments are built and cached in the fragment cache. When the application text was changed after a fragment had been cached, Strata would not recognize the change, and would continue to execute the cached fragment. As a result, when the test case was run under Strata, the string "Hello world" was output to the screen twice.

### 3. Overview of Approach

For Strata to handle the test case, it must retranslate instructions whenever executable code in the application is changed. It was therefore necessary to develop a mechanism for notifying Strata when an instruction is modified. This notification is accomplished by using memory protection and signal delivery mechanisms. The retranslation of instructions is enforced in the sequence of steps shown in Figure 3 and discussed in the following sections.

#### During Initialization

1)**Turn off write permissions.** The write permissions to executable code are turned off in order to catch attempted self-modifications.

2) **Set up mprotect interceptor.** An mprotect interceptor is set up to intercept all mprotect system calls to ensure that any attempt to make an executable page writable is unsuccessful. This step causes a segmentation fault to occur when the application attempts to modify executable code.

3) **Set up signal handler.** A signal handler is set up to intercept the SIGSEGV signal delivered by the segmentation fault, and to replace the signal's default action with signal handler code.

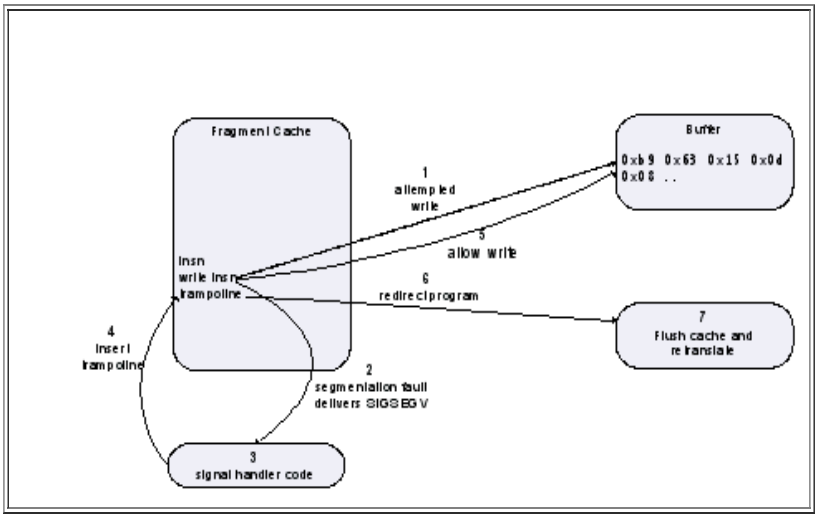


**Figure 3: Modifying Strata to Handle the Test Case**

#### During Execution

1) **Invoke signal handler.** When a write to a write-protected executable page is attempted (Step 1 of Figure 3), a segmentation fault occurs and causes a SIGSEGV signal to be delivered, shown in Step 2 of the figure. This signal is intercepted by the signal handler set up during initialization.

2) **Turn on write permissions.** When the signal handler code is invoked (Step 3 of Figure 3), the target page is made writable so that the write can occur when Strata returns from the fault.

3) **Insert trampoline.** The signal handler also inserts a trampoline after the fault-causing instruction (i.e., the write) in the fragment cache, as shown in Step 4 of Figure 3. Inserting the trampoline requires the computation of the size of the instruction, as well as the application address that corresponds to the fragment PC of the fault-causing instruction. The final action of the trampoline is a jump to cache-flushing code in Strata.

4) **Return from fault.** The signal handler code returns, and the execution of the application continues at the fault-causing write. The write is successfully made (Step 5 of Figure 3), followed by the execution of the trampoline.

5) **Flush cache and retranslate.** When the trampoline is executed, in Step 6 of Figure 3, it redirects the program to code that flushes the fragment cache so that the modified application instructions can be retranslated and placed in the fragment cache (Step 7 of Figure 3). Instructions are retranslated starting immediately after the write instruction in order to ensure that, if the instruction following the modification is affected, this change will be reflected when the instruction is executed.

### 4. Results and Contributions

To evaluate Strata's ability to handle self-modifying code in a general way, it was necessary to find real programs that make frequent use of this programming technique. The Low-Level Virtual Machine (LLVM) is a compiler infrastructure developed at the University of Illinois at Urbana-Champaign [6]. LLVM is made up of several components, including a GCC-based C and C++ front-end, a link-time optimization framework with a growing set of global and interprocedural analyses and transformations, static back-ends for x86, x86-64, PowerPC 32/64, ARM, Thumb, IA-64, Alpha and SPARC architectures, a back-end which emits portable C code and a Just-In-Time (JIT) compiler for x86, x86-64, and PowerPC 32/64 processors. The LLVM infrastructure was chosen as a test case for Strata's handling of self-modifying code because of its relatively small size, and the presence of the JIT compiler which routinely uses self-modifying code.

LLI, the JIT compiler in LLVM, works by mmap-ping several pages for the explicit purpose of code generation and modification. These pages are given read, write and execute permissions, so that code can be written to and executed from them. Under Strata's control, the self-modifying code handling mechanisms turn off the write permissions on these pages when instructions are fetched from them, and then turn them back on when a segmentation fault caused by an attempted write to a page is seen and handled.

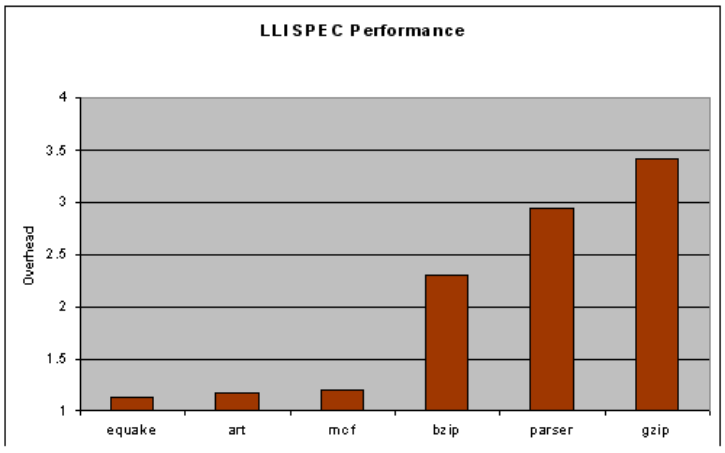


Figure 4: SPEC 2000 Runtimes

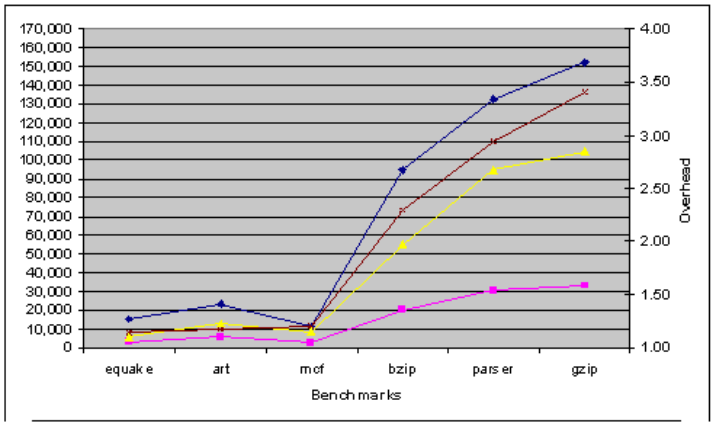


Figure 5: Relationship Between Self-Modifying Code and Strata Overhead

### Results

Figure 4 shows the performance of LLI on a selection of the SPEC 2000 benchmarks when run under the control of Strata, normalized to native execution of LLI. There is a slowdown seen in every case, ranging from 1.14X in the case of equake to 3.41X in the case of gzip. Figure 5 further explains the relationship between the amount of self-modifying code in an application and Strata's performance. In the figure, the number of bytes of machine code compiled, the number of x86 machine instructions emitted, and the number of fragment cache flushes performed by Strata are compared on the left-hand scale, and the overhead of running LLI under Strata is plotted on the right-hand scale of the graph. Figure 5 shows that the number of fragment cache flushes is proportional to the amount of jitted code and emitted machine instructions in the application, and that overhead is in turn proportional to fragment cache flushes. These results are not unexpected as the semantics of ensuring correctness when handling self-modifying code require flushing the fragment cache

after every instruction modification. Since flushing is a high-cost operation, applications with a great deal of self-modification would suffer degraded performance. Future efforts will focus on reducing the slowdown caused by Strata's self-modifying code handling mechanisms.

## Conclusions and Future Work

As mentioned in the previous section, we plan to improve the performance of Strata's self-modifying code handling mechanisms. We would like to investigate how more intelligent methods (such as flushing the cache after a certain number of modified instructions, or the use of an API to designate self-modifying regions within the application) would improve performance. In addition, as implemented, Strata allows self-modifying code to occur without any restrictions, as long as the target page has previously been made writable. We plan to optimize this work by getting rid of the assumption that all such write attempts should be allowed. This would involve including a verification process to verify whether or not an attempted write is valid. In the future, we hope to be able to use Strata as a security measure against malicious uses of self-modifying code, such as in metamorphic viruses.

### References


[1] Aycock, J. A Brief History of Just-In-Time. In ACM Computing Surveys, Volume 35, Issue 2, pp. 97-113, June 2003

[2] Bruening, D., Amarasinghe, S. Maintaining Consistency and Bounding Capacity of Software Code Caches. In Proceedings of the International Symposium on Code Generation and Optimization, pp. 74-85, March 2005

[3] Hiser, J.D., Williams, D., Filipi, A., Davidson, J.W., Childers, B.R. Evaluating Fragment Construction Policies for SDT Systems. In Proceedings of the Second International Conference on Virtual Execution Environments, pp. 2-12, June 2006

[4] Hiser, J.D., Williams, D., Hu, W., Davidson, J.W., Mars, J., Childers, B.R. Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems. In International Symposium on Code Generation and Optimization, March 2007

[5] Hu, W., Hiser, J., Williams, D., Filipi, A. Davidson, J.W., Evans, D., Knight, J.C., Nguyen-Tuong, A., Rowanhill, J. Secure and Practical Defense Against Code-Injection Attacks using Software Dynamic Translation. In Proceedings of the Second International Conference on Virtual Execution Environments, pp. 2-12, June 2006

[6] Lattner, C., Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the International Symposium on Coge Generation and Optimization , pp. 75-86, March 2004

[7] Scott K., Kumar, N., Velusamy, S., Childers, B., Davidson, J.W., Soffa, M.L. Retargetable and Reconfigurable Software Dynamic Translation. In Proceedings of the International Symposium on Code Generation and Optimization, pp. 36-47, March 2003

[8] Szor, P., Ferrie, P. Hunting for Metamorphic. In Proceedings of the Virus Bulletin Conference, September 2001