Safe Virtual Execution Using Software Dynamic Translation

Kevin Scott and Jack Davidson

Department of Computer Science, University of Virginia
Charlottesville, VA 22904
{kscott, jwd}@cs.virginia.edu

Abstract

Safe virtual execution (SVE) allows a host computer system to reduce the risks associated with running untrusted programs. SVE prevents untrusted programs from directly accessing system resources, thereby giving the host the ability to control how individual resources may be used. SVE is used in a variety of safety-conscious software systems, including the Java Virtual Machine (JVM), software fault isolation (SFI), system call interposition layers, and execution monitors. While SVE is the conceptual foundation for these systems, each uses a different implementation technology. The lack of a unifying framework for building SVE systems results in a variety of problems: many useful SVE systems are not portable and therefore are usable only on a limited number of platforms; code reuse among different SVE systems is often difficult or impossible; and building SVE systems from scratch can be both time consuming and error prone.

To address these concerns, we have developed a portable, extensible framework for constructing SVE systems. Our framework, called Strata, is based on software dynamic translation (SDT), a technique for modifying binary programs as they execute. Strata is designed to be ported easily to new platforms and to date has been targeted to SPARC/Solaris, x86/Linux, and MIPS/IRIX. This portability ensures that SVE applications implemented in Strata are available to a wide variety of host systems. Strata also affords the opportunity for code reuse among different SVE applications by establishing a common implementation framework.

Strata implements a basic safe virtual execution engine using SDT. The base functionality supplied by this engine is easily extended to implement specific SVE systems. In this paper we describe the organization of Strata and demonstrate its extension by building two SVE systems: system call interposition and stack-smashing prevention. To illustrate the use of the system call interposition extensions, the paper presents implementations of several useful security policies.

1. Introduction

Today's software environment is complex. End users acquire software from a number of sources, including the network, and have very little on which to base their trust that the software will correctly perform its intended function. Given the size of modern software—operating system kernels are comprised of millions of lines of source code and application programs are often an order of magnitude larger—it is difficult or impossible for developers to guarantee that their software is worthy of the end user's trust. Even if developers could make such guarantees about the software they distribute, hostile entities actively seek to modify that software to perform unanticipated, often harmful functions via viruses and Trojan horses.

In recent years, researchers have developed a variety of techniques for managing the execution of untrusted code. These techniques can be divided into two orthogonal categories: static and dynamic. Static techniques analyze untrusted binaries before execution to determine whether or not the program is safe to run. Proof carrying code [17] is a good example of the static approach—before a program can execute, the runtime system must successfully validate a proof that the untrusted binary will adhere to a given safety policy. Many static approaches, including proof carrying code, rely on source code analyses to produce safe binaries [5,15,22]. Dynamic techniques, on the other hand, do not require access to source code. Rather, dynamic techniques prevent violation of safety policies by monitoring and modifying the behavior of untrusted binaries as they execute. An example of a dynamic approach is execution monitoring [9,18]. Execution monitors terminate the execution of a program as soon as an impermissible sequence of events (corresponding to a safety policy violation) is observed. System call interposition layers [11, 12, 13, 14] are similar to execution monitors with the additional ability to alter the semantics of events, specifically system calls. Yet another similar dynamic technique, software fault isolation (also

known as sandboxing) [23] limits the potential damage an untrusted binary can do by preventing loads, stores, or jumps outside of a restricted address range.

In this paper we make the following observation: many dynamic trust management systems, including the ones mentioned above, can be implemented using a technique called safe virtual execution (SVE). SVE mediates application execution, virtualizing access to sensitive resources in order to prevent untrusted binaries from causing harm. Despite the fact that SVE provides a conceptual framework for the implementation of systems such as execution monitors, interposition layers, and sandboxing, these systems are frequently based on widely differing implementation technologies. These systems are often dependent on a specific target architecture or on special operating system services, hence impeding their widespread use in the modern heterogeneous networked computing environment. In addition to non-portability, the use of different implementation technology places undue engineering burdens on the designers of SVE systems. They cannot share code and features with similar systems and must often endure the time consuming and error-prone chore of building their systems from scratch.

To address these concerns, we have developed a portable, extensible framework for constructing SVE systems. Our framework, called Strata, is based on software dynamic translation (SDT), a technique for modifying binary programs as they execute [1, 2, 3, 6, 7, 20, 21, 24]. Using SDT, Strata offers a basic safe virtual execution engine. The base functionality supplied by this engine can be extended in order to implement specific SVE systems. Using this approach useful SVE systems can often be implemented with very few lines of new code. Strata is designed to be easily ported to new platforms and to date has been targeted to SPARC/Solaris, x86/Linux, and MIPS/IRIX. This portability ensures that SVE applications implemented in Strata are available to a wide variety of host systems. Strata also affords the opportunity for code reuse among different SVE applications by establishing a common implementation framework.

The remainder of this paper is organized as follows. Section 2 provides an overview of software dynamic translation and Section 3 describes Strata's organization and architecture. Section 4 then describes how Strata is used to implement a system call interposition layer and how this layer can be used to implement powerful security policies. Section 5 discusses our results while Section 6 discusses related work, and Section 7 provides a summary.

2. Software Dynamic Translation

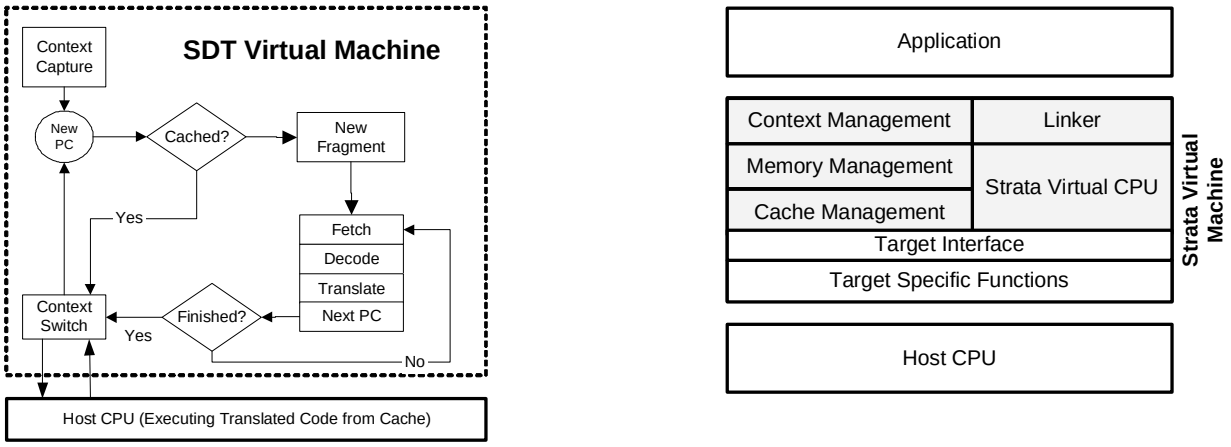
SDT is a technique for dynamically modifying a program as it is being executed. Software dynamic translation has been used in a variety of different areas: binary translation for executing programs on non-native CPUs [6, 7, 21]; fast machine simulation [3, 24]; and recently, dynamic optimization [1]. In this paper we describe how software dynamic translation can be used to implement safe virtual execution.

Most software dynamic translators are organized as virtual machines (see Figure 1a). The virtual machine fetches instructions, performs an application-specific translation to native instructions, and then arranges for the translated instructions to be executed. Safe virtual execution systems can be viewed as types of virtual machines. On a conceptual level, an SVE virtual machine prevents untrusted binaries from directly manipulating system resources. The difference between SVE systems is in how this virtual machine is implemented. For instance, in the Java Virtual Machine an interpreter is used to isolate Java bytecode programs from underlying system resources [16]. Systems such as SASI [9] and SFI [23] merge the application program with the SVE virtual machine, using binary rewriting at load time; the virtual machine is in the form of instructions that check certain sequences of instructions before they are allowed to execute. Systems such as Janus [13] and Interposition Agents [14] use special operating system facilities to virtualize the execution of a very specific aspect of execution, specifically, system calls.

In this paper we propose the use of software dynamic translation as the basis for implementing safe virtual execution systems. Implementing an SVE application in a software dynamic translator is a simple matter of overriding the translator's default behavior. For example, an SDT implementation of a software fault isolator would translate load instructions into a sequence of instructions that performs an address check before the load executes.

In order to illustrate our approach in brief, consider the task of preventing stack-smashing attacks using SDT. Stack-smashing attacks take advantage of unsafe buffer manipulation functions (e.g., `strcpy` from the C standard library) to copy, and subsequently execute, malicious code from the application stack. The malicious code is executed with the privileges of the user running the program, and in many cases can lead to serious security compromises on affected systems [4,15].

A simple way to prevent stack-smashing attacks is to make the application stack non-executable. In the abscence of operating system support for non-executable stacks, it is a trivial matter to prevent execution of



(a) (b)

Figure 1: Strata Architecture

code on the stack by using SDT. This task is accomplished by replacing the software dynamic translator's default fetch function with a custom fetch that prevents execution of stack resident code.
The custom fetch function

```
custom_fetch (Address PC) {
   if (is_on_stack(PC)) {
       fail("Cannot execute code on the
stack");
   } else {
       return default_fetch(PC);
   }
}
```

checks the PC against the stack boundaries and terminates program execution if the instruction being fetched is on the stack. If the instruction being fetched is not on the stack, it is alright to execute the instruction, and consequently the fetch is completed by calling the default fetch function.

3. Strata

To facilitate SDT research and the development of innovative SDT applications, we have constructed a portable, extensible SDT infrastructure called Strata. As shown in Figure 1a, Strata is organized as a virtual machine. The Strata VM mediates application execution by examining and translating instructions before they execute on the host CPU. Translated instructions are held in a Strata-managed cache. The Strata VM is entered by capturing and saving the application context (e.g., PC, condition codes, registers, etc.). Following context capture, the VM processes the next application instruction. If a translation for this instruction has been cached, a *context switch* restores the application context and begins executing cached translated instructions on the host CPU.

If there is no cached translation for the next application instruction, the Strata VM allocates storage for a new *fragment* of translated instructions. A fragment is a sequence of code in which branches may appear only at the end. The Strata VM then populates the fragment by fetching, decoding, and translating application instructions one-by-one until an end-of-fragment condition is met. The end-of-fragment condition is dependent on the particular software dynamic translator being implemented. For many translators, the end-of-fragment condition is met when an application branch instruction is encountered. Other translators may form fragments that emulate only a single application instruction. In any case, when the end-of-fragment condition is met, a *context switch* restores the application context and the newly translated fragment is executed.

As the application executes under Strata control, more and more of the application's working set of instructions materialize in the fragment cache. This, along with certain other techniques—e.g., partial inlining of functions and indirect branch elimination—that reduce the number and cost of context switches, permits Strata to execute applications with little or no measurable overhead [19].

Figure 1b shows the components of the Strata VM. Strata was designed with extensibility and portability in mind. Extensibility allows Strata to be used for a variety of different purposes; researchers can use Strata to build dynamic optimizers, dynamic binary translators, fast architecture emulators, as well as safe virtual execution systems. Portability allows Strata to be moved to new machines easily. To date, Strata has been ported to SPARC/Solaris, x86/Linux, and MIPS/IRIX. More

importantly, Strata's portability means that software implemented using Strata's extensibility features is readily available on a wide range of target architectures and operating systems.

To achieve these goals, the Strata virtual machine is implemented as a set of target-independent *common services*, a set of *target-specific functions*, and a reconfigurable *target interface* through which the machine-independent and machine-dependent components communicate (see Figure 1b). Implementing a new software dynamic translator often requires only a small amount of coding and a simple reconfiguration of the target interface. Even when the implementation is more involved, e.g., when retargeting the VM to a new platform, the programmer is only obligated to implement the target-specific functions required by the target interface; common services should never have to be reimplemented or modified.

Strata consists of 5000 lines of C code, roughly half of which is target-specific. In Figure 1b, shaded boxes show the Strata common services which comprise the remaining half of the Strata source. The Strata common services are target-independent and implement functions that may be useful in a variety of Strata-based dynamic translators. Features such as context management, memory management, and the Strata virtual CPU will most likely be required by any Strata-based dynamic translator. The cache manager and the linker can be used to improve the performance of Strata-based dynamic translators, and are detailed in other work [19].

4. Strata and Safe Virtual Execution

In Section 2 we sketched one example that demonstrates the process one can use to write a Strata-based safe virtual execution system, specifically, a stack-smashing inhibitor. In this section we use Strata to implement a system call interposition layer. This interposition layer, like all Strata-based applications, is user-level software and requires no kernel modifications. Our Strata-based system call interposition layer also obviates the need for special operating system services for interception or redirection of system calls. As a consequence, our system call interposition layer is more flexible and portable than many existing systems.

SDT's ability to control and dynamically modify a running program provides an ideal mechanism for implementing a system call interposition layer. As the untrusted binary is virtualized and executed by Strata, code is dynamically inserted to intercept system calls and potentially redirect those calls to user supplied functions. In general though, this process does not need to be limited to system calls; all access to host CPU and operating system resources are explicitly controlled by Strata (see Figure 2).

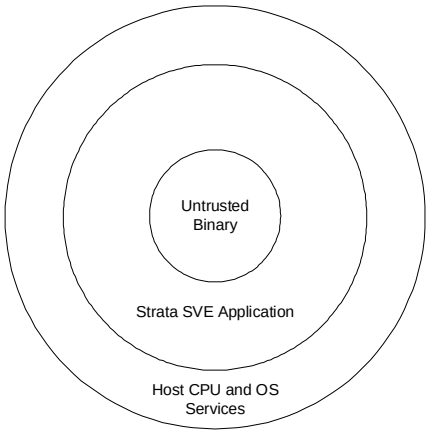


Figure 2: Strata

In this paper, we will use terms and phrases that are typically employed when discussing the Unix operating system (e.g., "becoming root", "exec'ing a shell", "performing a setuid(0)", etc.). The actions indicated by these terms have analogs in other major operating systems (e.g., Windows NT, Windows 2000, Window XP, VxWorks, and PSOSystem) and the approaches we describe would apply equally well to applications running on these systems.

A simple, but realistic example illustrates our approach. Suppose a user wishes to enforce a policy that prohibits untrusted applications from reading a file that the user normally has permission to read. Let's call this file `/etc/passwd` (`registry.dat`, `SAM`, or `system` might be equally good choices). Now assume that the user receives an untrusted binary called `funny` and wishes to run it. The user invokes `funny` using the Strata loader. The Strata loader locates the entry point of the application and inserts a call to the Strata startup routine. When the loader begins the execution of the application, the call to the Strata startup routine leads to the dynamic loading and invocation of Strata.

As Strata processes `funny`'s text segment and builds fragments to be executed, it locates *open* system calls and replaces them with code that invokes the execution steering policy code. When the fragment code is executed, all *open* system calls are diverted to the policy code. It is the policy code's job to examine the arguments to the original *open* system call. If the untrusted application is attempting to open `/etc/passwd`, an error message is issued and the execution of the application is terminated. If the file being opened is not `/etc/passwd`, the security policy code performs the

open request, returns the result, and execution continues normally (albeit under the control of Strata).

4.1. A System Call Interposition API

We support system call interposition through an API implemented by overriding Strata's base functionality. The API is a simple, efficient mechanism that allows the user to specify which operating system calls are to be monitored and the code to execute every time the operating system call is invoked. Strata's execution steering API consists of four functions. They are:

```
void init_syscall();
watch_syscall(unsigned num, void *callback);
void strata_policy_begin(unsigned num);
void strata_policy_end(unsigned num);
```

The first function is called on the initial entry to Strata. The implementation of this function will contain calls to the second API function `watch_syscall()`. Function `watch_syscall()` specifies an operating system call to watch (i.e., `num`) and the redirected system call to execute when that OS call is invoked (i.e., `callback`). The signature of `callback` should match the signature of the operating system call being watched. The final two API functions are used to bracket redirected system call code. The need for the bracketing functions will be explained shortly when we describe how Strata dynamically injects code into the application.

To illustrate the implementation of Strata's security API, we show the Strata security policy for preventing an untrusted application from reading `/etc/passwd`. Following the style used on hacker websites to demonstrate the exploitation of security vulnerabilities, we give a small demonstration program that exercises the policy. The demonstration code is given in Listing 1.

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <strata.h>
4.  #include <sys/syscall.h>
5.  int myopen (const char *path, int oflag) {
6.      char absfilename[1024];
7.      int fd;
8.      strata_policy_begin(SYS_open);
9.      makepath_absolute(absfilename,path,1024);
10.     if (strcmp(absfilename,"/etc/passwd") == 0) {
11.         strata_fatal("Naughty, naughty!");
12.     }
13.     fd = syscall(SYS_open, path, oflag);
14.     strata_policy_end(SYS_open);
15.     return fd;
16. }
17. void init_syscall() {
18.     (*TI.watch_syscall)(SYS_open, myopen);
19. }
```

Listing 1: Code for preventing a file from being opened.

```
20.
21. int main(int argc, char *argv[]) {
22.     FILE *f;
23.     if (argc < 2 || (f = fopen(argv[1],"r")) ==
    NULL) {
24.         fprintf(stderr,"Can't open file.\n");
25.         exit(1);
26.     }
27.     printf("File %s opened.\n",argv[1]);
28.     return 0;
29. }
```

Listing 1: Code for preventing a file from being opened.

Before explaining how Strata injects this code into an untrusted binary, we review the code at a high level. Function `init_syscall()` at lines 17–19 specifies that `SYS_open` calls should be monitored and that when a `SYS_open` call is to be executed by the application, control is to be transferred to the policy routine `myopen()`.

Function `myopen()` (lines 5–16) implements the redirected system call. As mentioned previously, invocations of `strata_policy_begin()` and `strata_policy_end()` are used to bracket the redirected system call code and their purpose will be explained shortly.

In function `myopen()`, the path to be opened is converted to an absolute pathname by calling the utility function `makepath_absolute()`. The path returned is compared to the string `/etc/passwd` and if it matches, an error message is issued and execution is terminated. If the file to be opened is not `/etc/passwd`, then the policy code performs the `SYS_open` system call and returns the result to the client application as if the actual system call was executed.

When an untrusted binary is to be executed, the Strata loader modifies the application binary so that initial control is transferred to Strata's initialization routines. This routine dynamically loads and executes the `init_syscall()` function that sets up a table of system calls to watch and their corresponding callback functions.

After initialization is complete, Strata begins building the initial application fragment by fetching, decoding and translating instructions from the application text into the fragment cache. The system call interposition API is implemented by overriding the translate function that handles trap or interrupt instructions. For the SPARC/Solaris platform, less than 20 lines of code are required to implement the new translation functionality.

Strata examines each operating system call site to determine if the OS call is one to be monitored. In most cases, Strata can determine at translation time which operating system call will be invoked at the call site. If

the OS call is one to be monitored, the code to invoke the operating system call is replaced with a call to the user-supplied code. If the call is not one to be monitored, no translation action needs to be taken and the operating system call code is copied unchanged to the fragment cache.

In some cases, Strata cannot determine which operating system call will be invoked at a given call site. This can occur, for instance, with indirect operating system calls. In these cases, Strata must generate and insert code that, when the fragment is executed, will test whether the OS call being invoked is one to be monitored. If the call is one to be monitored, the inserted code must call the appropriate user-supplied policy code; otherwise, the OS call is executed.

In the case where the OS call to be invoked can be determined at fragment creation (translation) time, Strata treats redirected code just like application code. As a result, calls to redirected system call code can often be partially inlined [1, 19], thus improving the efficiency of the code. However, partially inlining code creates a complication. Consider the `myopen()` code in Listing 1. When this code is inlined, the `SYS_open` OS call will be generated. This OS call should not be replaced by a callback, as it is the OS call to execute when the policy's conditions are satisfied. To avoid infinite recursion, redirected system calls are bracketed using the interposition API calls `strata_policy_begin()` and `strata_policy_end()`. Strata uses these "code markers" to suspend the translation of operating system calls. Thus, we are assuming that the writer of policy code is not malicious.

One further complication exists. A malicious user with knowledge of how Strata operates may try to circumvent Strata by using calls to `strata_policy_begin()` and `strata_policy_end()` to bracket application code that attempts to violate the security policy. To prevent this avenue of attack, Strata permits `strata_policy_begin()` and `strata_policy_end()` to execute only from within security policy code.

4.2. System Call Interposition at Work

A common security exploit is to arrange to exec a shell while in root or super-user mode. This is most commonly done by using a buffer overrun attack that corrupts the run-time stack. In an earlier paper we described how such an attack can be stopped using Strata's target-dependent interfaces [13]. Other types of attacks are possible [16]. However, they all rely on exec'ing a program (usually a shell) while in root or super-user mode. Using Strata's security API, it is very simple to write a policy that prohibits exec'ing a program when in super-user mode, yet allows exec's when not in super-user mode. Listing 2 contains the demonstration program.

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <unistd.h>
4.  #include <strata.h>
5.  #include <sys/syscall.h>
6.  static int curuid = -1;
7.  int mysetuid (int uid) {
8.      strata_policy_begin(SYS_setuid);
9.      curuid = syscall(SYS_setuid, uid);
10.     strata_policy_end(SYS_setuid);
11.     return curuid;
12. }
13. int myexecve (const char *path, char *const
    argv[],
14.  char *const envp[]) {
15.     int retval;
16.     strata_syscallback_begin(SYS_execve);
17.     if (curuid == 0)
18.         strata_fatal("Naughty, naughty");
19.     retval = syscall(SYS_execve, path, argv,
    envp);
20.     strata_syscallback_end(SYS_execve);
21.     return retval;
22. }
23. void init_syscall() {
24.     (*TI.watch_syscall)(SYS_execve, myexecve);
25.     (*TI.watch_syscall)(SYS_setuid, mysetuid);
26. }
27. int main (int argc, char *argv[]) {
28.     FILE *f;
29.     char *args[2] = {"/bin/sh",0};
30.     setuid(0);
31.     execv("/bin/sh", args);
32.     return 0;
33. }
```

Listing 2: Code to prevent exec's while root.

Two system calls—`setuid` and `execve`—must be monitored to implement this security policy. We must monitor `setuid` to keep track of the uid of the running application. This information is stored in the state variable `curuid`. In function `myexecve()`, exec's are disallowd if the program is running in root mode (i.e., the uid of the process is 0); otherwise they are allowed.

This example demonstrates a number of advantages of our system call interposition API. It is easy to see that the code required to implement the security policy using the system call interposition API is simple and straightforward. We do not rely on special operating system services, compilers, or libraries. The user does not have to learn a new domain specific language in order to write security policies. Furthermore, the use of C as the security policy language does not imply that untrusted binaries must be written in C. The security policy is compiled to binary code that is processed by

Strata along with the untrusted binary. The security policy code is portable to most systems with a native C compiler and POSIX compliant system calls. Moreover, static source code analysis cannot effectively prevent execs while root due to a number of inhibiting factors—unavailability of library source code, dynamically generated code, self-modifying code, and the inability of static analyses to precisely predict dynamic state.

The third security policy presented implements a policy that controls the rate at which an application uses a resource. In this example, we will limit the rate at which an application can transmit packets over a socket. This type of policy could be useful for thwarting denial of service attacks where zombie processes attempt to flood a server with packets. Listing 3 gives the code for the demonstration application.

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <sys/types.h>
4.  #include <sys/socket.h>
5.  #include <netinet/in.h>
6.  #include <netdb.h>
7.  #include <time.h>
8.  #include <string.h>
9.  #include <strata.h>
10. #include <sys/syscall.h>
11. #define RATE 10000
12. #define TOPRATE 10000000
13. #define DISCARD_PORT 9999
14. #define PAYLOAD_SIZE 1024
15. void xmit (const char *host, int nbytes);
16. static int socket_fd = -1;
17. /* Compute the delay necessary to maintain */
18. /* the desired rate */
19. int limiting_delay (double rate, time_t tbeg,
20.  time_t tend, int last_len, int len);
21. /* Callback for the so_socket call */
22. int my_so_socket (int a,int b,int c,char *d,int e)
    {
23.     strata_policy_begin(SYS_so_socket);
24.     /* Make the system call and */
25.     /* record the file descriptor */
26.     socket_fd = syscall(SYS_so_socket,a,b,c,d,e);
27.     strata_policy_end(SYS_so_socket);
28.     return socket_fd;
29. }
30. /* Callback for the write system call */
31. int my_send (int s, const void *msg, size_t len,
32.  int flags) {
33.     int result;
34.     time_t now;
35.     static int last_len = 0;
36.     static time_t last_time = 0;
37.     strata_policy_begin(SYS_send);
38.     /* Only look at writes to socket_fd */
39.     if (s == socket_fd) {
40.         now = time(NULL);
41.         sleep(limiting_delay(RATE,last_time, now
42.          len,last_len));
```

Listing 3: Code to limit the rate of transmission over a socket.

```
43.         last_len = len;
44.         last_time = now;
45.     }
46.     result = syscall(SYS_send,s,msg,len,flags);
47.     strata_policy_end(SYS_send);
48.     return result;
49. }
50. void init_syscall() {
51.
    (*TI.watch_syscall)(SYS_so_socket,my_so_socket);
52.     (*TI.watch_syscall)(SYS_send,my_send);
53. }
54. main(int argc, char *argv[]) {
55.     if (argc == 3)
56.         xmit(argv[1],atoi(argv[2]));
57.     else
58.         fprintf(stderr,
59.          "Usage: %s host nbytes\n",argv[0]);
60. }
61. /* Transmit nbytes to discard port (9) on host */
62. void xmit (const char *host, int nbytes) {
63.     int sd, bytes_sent;
64.     struct sockaddr_in sin;
65.     struct sockaddr_in pin;
66.     struct hostent *hp;
67.     char *payload[PAYLOAD_SIZE];
68.     time_t begin, elapsed;
69.     double rate;
70.     /* go find out about the desired host machine
    */
71.     if ((hp = gethostbyname(host)) == 0) {
72.         perror("gethostbyname");
73.         exit(1);
74.     }
75.     /* fill in the socket structure with host info
    */
76.     memset(&pin, 0, sizeof(pin));
77.     pin.sin_family = AF_INET;
78.     pin.sin_addr.s_addr = ((struct in_addr *)
79.          (hp->h_addr))->s_addr;
80.     pin.sin_port = htons(DISCARD_PORT);
81.     /* grab an Internet domain socket */
82.     if ((sd = socket(AF_INET,SOCK_STREAM, 0)) == -
    1) {
83.         perror("socket");
84.         exit(1);
85.         }
86.     /* connect to PORT on HOST */
87.     if (connect(sd, (struct sockaddr *) &pin,
88.          sizeof(pin)) == -1) {
89.         perror("connect");
90.         exit(1);
91.     }
92.     begin = time(0);
93.     bytes_sent = 0;
94.     while(bytes_sent < nbytes) {
95.         /* send a message to the server PORT */
96.         /* on machine HOST */
97.         if (send(sd,payload,sizeof(payload),0) == -
    1) {
98.             perror("send");
99.             exit(1);
100.        }
101.        bytes_sent += sizeof(payload);
102.        printf(".");
103.        fflush(stdout);
```

Listing 3: (Continued)Code to limit the rate of transmission over a socket.

```
104.    }
105.    elapsed = time(0) - begin;
106.    rate = bytes_sent / elapsed;
107.    printf("\nRate = %8.3f bytes per
    second.\n",rate);
108.    close(sd);
109.}
```

Listing 3: (Continued)Code to limit the rate of transmission over a socket.

To implement this policy, `SYS_so_socket` and `SYS_send` system calls must be monitored. Callbacks `SYS_so_socket (my_so_socket)` and `SYS_send (my_send)` are established (lines 50–53). The policy code for monitoring the socket call simply records the file descriptor for the socket. The recorded file descriptor will be used by `my_send()` to limit the rate only on this connection. In function `my_send()`, if the transmission is to the monitored connection (i.e., `socket_fd`), then a delay is introduced when necessary (see line 41 of Listing 3).

Listing 4 contains our fourth and final Strata security demonstration program. The security policy prevents cookies from being transmitted to web servers. In this example, the two system calls to be monitored are `SYS_so_socket` and `SYS_write`. Like the previous example, the callback `my_so_socket()` simply remembers the socket being opened. In callback `my_write()`, writes to the socket are detected and the buffer is preprocessed by `remove_cookies()` before writing it (see lines 29–33 of Listing 4)..

Whereas the second security policy (Listing 2) halts programs that violate a given system call utilization pattern, the final two policies permit programs to continue running with altered system call semantics. The third security policy (Listing 3), in effect, can prevent untrusted binaries from mounting an effective network denial of service attack from a Strata protected host. The third security policy can help a user maintain his/her privacy while navigating the web.

5. Discussion

As the previous section has shown, writing powerful software security policies using Strata's security API is simple. We implemented the API under SPARC/Solaris by adding fewer than 20 lines of code to the base Strata system. The implementation of the interposition API for other Strata supported architectures (x86/Linux, MIPS/IRIX) would also require relatively few lines of code. Developing a system call interposition system from scratch, even if using non-standard operating system facilities, would on the other hand require a much greater implementation effort.

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <strata.h>
4.  #include <sys/syscall.h>
5.  #include "snarf.h"
6.
7.  static int socket_fd = -1;
8.
9.  /* Copy src buffer to dst removing cookies*/
10. int remove_cookies(char *dst, const void *src,
11.  int size);
12. /* Callback for the so_socket system call. */
13. int my_so_socket (int a, int b, int c, char *d,
14.  int e) {
15.     strata_policy_begin(SYS_so_socket);
16.     /* Make the system call and record the */
17.     /* file descriptor */
18.     socket_fd = syscall(SYS_so_socket,a,b,c,d,e);
19.     strata_policy_end(SYS_so_socket);
20.     return socket_fd;
21. }
22. /* Callback for the write system call */
23. int my_write (int fd, void *buf, int size) {
24.     char new_buf[1024];
25.     int s, new_size;
26.     strata_policy_begin(SYS_write);
27.     /* Only look at writes to socket_fd
28.     /* and only rewrite HTTP headers. */
29.     if (fd == socket_fd &&
30.      (new_size =
    remove_cookies(new_buf,buf,size)))
31.        s = syscall(SYS_write,fd,new_buf,new_size);
32.     else
33.        s = syscall(SYS_write,fd,buf,size);
34.     strata_policy_end(SYS_write);
35.     return s;
36. }
37. void init_syscall() {
38.
    (*TI.watch_syscall)(SYS_so_socket,my_so_socket);
39.     (*TI.watch_syscall)(SYS_write,my_write);
40. }
41. int main(int argc, char *argv[]) {
42.     snarf_main(argc, argv);
43. }
```

Listing 4: Code to remove cookies.

While it is somewhat ironic that we wrote our sample security policies using C (a cause of many security vulnerabilities in general) we did so to make the techniques and policies accessible to the largest audience. The policies presented here could have been written in any language. The only requirements are that bindings of Strata's security API must be implementable in the new target language and that a compiler for the target language is available which emits object code. Providing a new language implementation of Strata's security API is easy since the API consists of four simple functions.

Our current implementation does, however, have a few limitations on the types of SVE systems that can be built. For instance, Strata does not currently handle

multi-threaded code. We are currently extending Strata so that SVE designers can build systems that handle threaded code. Nonetheless, we feel that our approach provides a useful and complementary approach to helping provide software security.

Another important issue to consider is the overhead of using SDT for SVE. High overheads will limit the applicability of using SDT in SVE systems. In a previous paper, we discussed techniques for reducing overhead and showed that SDT was competitive with previously developed techniques for preventing certain classes of security breaches [14]. Currently, the slowdown of running an application under Strata is negligible, but in some cases can be as much as 1.32X. For many types of applications, 30 percent overhead is acceptable. Examples include executing an e-mail attachment which includes a self-extracting archive, opening a foreign document that contains malicious macros which destroy valuable information, and many setuid programs which perform simple administrative functions. For these situations, a slowdown of 20 to 30 percent would not be noticeable to the user.

For some applications such as web servers, web browsers, and databases, an overhead of 30 percent might not be acceptable. Indeed for these types of applications, any overhead is unlikely to be acceptable. Fortunately, previous research on dynamic optimization has shown that it is possible to achieve substantial speedups in long running applications [1, 2, 11]. Thus we believe that by combining a dynamic security checker with a dynamic optimizer, CPU-intensive applications can be run securely without overhead. To this end, we are working to incorporate additional optimizations within Strata's framework. If successful, our approach to SVE implementation would be applicable to an even wider range of applications.

6. Related Work

Many researchers have proposed and implemented system call interposition layers. Mike Jones proposed interposition agents an object-oriented, high-level framework for building interposition layers [14]. Interposition agents are portable as long as the host operating system provides a system call interception and redirection interface. Ghormley, et. al., proposed an interposition system based on a non-standard, reconfigurable, kernel-resident system call redirection API [12]. Fraser, et.al., have proposed a similar interposition system based on kernel loadable modules [11].

Considerable research has been conducted on execution monitoring [18], a technique similar to interposition layers. The Janus system monitors system calls executed by a program in order to determine whether or not a security policy has been violated [13]. Janus uses the operating system ptrace facility to register callbacks to policy enforcement code. Not all operating systems have such a facility, precluding Janus's use on those platforms. Janus also refrains from monitoring frequently executed system calls (e.g., write()) in order to keep overhead low. The SASI system, like Janus, also performs execution monitoring [9]. Rather than relying on an operating system facility, or restricting itself to monitoring only system calls, SASI inserts monitoring code required by the security policy directly into the program binary before execution. Consequently SASI cannot enforce the security policy on self-modifying code or dynamically generated code.

There are a variety of tools which dynamically prevent buffer overflow attacks using specialized code generation and/or binary rewriting techniques. The StackGuard system is a customized C compiler that generates code to dynamically detect and prevent the occurrence of most stack buffer overflows (i.e., a "stack-smashing" attack) [4]. The libverify tool uses a combination of late program modification and techniques borrowed from StackGuard to prevent buffer overflows, but without requiring a special compiler or access to source code [3]. While both of these tools are very useful, they are restricted to one specific safe virtual execution task, one programming language (C), and in the case of StackGuard, require a special compiler as well as special versions of standard libraries.

In addition to the dynamic execution management techniques discussed in this paper, there are also a variety of static techniques for improving the trustworthiness of programs. As mentioned earlier, proof carrying code systems [17] prevent untrusted binaries from ever executing if their trustworthiness cannot be statically proven. When source code is available, a variety of static analysis techniques can be used to locate buffer overflow vulnerabilities [10, 15], API misusages [5], and other programming errors that may result in security vulnerabilities [8, 22]. These techniques, and many other static source code analyses, are complementary to the dynamic techniques discussed in this paper.

7. Summary

In this paper we have shown that powerful safe virtual execution systems can be easily constructed using SDT and the Strata framework. Some SVE applications, such as stack-smashing prevention can be implemented in relatively few lines of code and are portable across any of Strata's supported architectures. Other SVE applications, such as system call interposition, require on the order of twenty lines of code per target architecture. Implementing either of these SVE systems

from scratch would require significantly greater implementation efforts, thus making Strata a particularly attractive starting point for SVE developers.

We have also shown several applications of our system call interposition API. These applications perform useful tasks, such as preventing the execution of root shells, limiting the effectiveness of network denial of service tasks, and helping users maintain the privacy of personal information. Moreover each of these applications were relatively simple to implement using Strata.

Given Strata's portability, extensibility, and the demonstrated ease with which it can be used to develop powerful SVE systems, we are confident that SDT and Strata will find use in the development of novel SVE systems.

8. Acknowledgements

This work was supported in part by an Intel Foundation Graduate Fellowship and by NSF grant EIA-0072043. We would like to thank Siva Velusamy for his work on Strata/x86 and Naveen Kumar and Bruce Childers for their work on Strata/MIPS. Lastly we would like to thank Shannon Hunt, whose careful reading of this paper greatly improved its quality.

9. References

[1] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation* (2000), pp. 1–12.

[2] CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. FX!32: A profile-directed binary translator. *IEEE Micro 18*, 2 (Mar.slash Apr. 1998), 56–64. Presented at Hot Chips IX, Stanford University, Stanford, California, August 24–26, 1997.

[3] CMELIK, B., AND KEPPEL, D. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems* (May 1994), pp. 128–137.

[4] COWAN, C., PU, C., MAIER, D., HINTON, H., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., , AND ZHANG, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 1998 USENIX Security Symposium* (1998).

[5] DELINE, R., AND FÄHNDRICH, M. Enforcing High-Level protocols in Low-Level software. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)* (N.Y., June 20–22 2001), C. Norris and J. J. B. Fenwick, Eds., vol. 36.5 of *ACM SIGPLAN Notices*, ACMPress, pp. 59–69.

[6] DITZEL, D. R. Transmeta's Crusoe: Cool chips for mobile computing. In *Hot Chips 12: Stanford University, Stanford, California, August 13–15, 2000* (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000), IEEE, Ed., IEEE Computer Society Press.

[7] EBCIOGLU, K., AND ALTMAN, E. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Annual International Symposium on Computer Architecture* (1997), pp. 26–37.

[8] ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation (OSDI 2000)* (San Diego, CA, 23–25 Oct. 2000).

[9] ERLINGSSON, Ú., AND SCHNEIDER, F. B. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop* (Caledon Hills, Ontario, Canada, Sept. 1999), ACM SIGSAC, ACM Press, pp. 87–95.

[10] EVANS, D., AND LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE Software 19*, 1 (Jan./ Feb. 2002), 42–51.

[11] FRASER, T., BADGER, L., AND FELDMAN, M. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy* (1999), pp. 2–16.

[12] GHORMLEY, D. P., PETROU, D., RODRIGUES, S. H., AND ANDERSON, T. E. SLIC: An extensibility system for commodity operating systems. pp. 39–52.

[13] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium* (1996).

[14] JONES, M. B. Interposition agents: Transparently interposing user code at the system interface. In *Symposium on Operating Systems Principles* (1993), pp. 80–93.

[15] LAROCHELLE, D., AND EVANS, D. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium* (2001).

[16] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*, 1 ed. Addison-Wesley, Reading/Massachusetts, 1996.

[17] NECULA, G. C. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France, jan 1997), pp. 106–119.

[18] SCHNEIDER, F. B. Enforceable security policies. *Information and System Security 3*, 1 (2000), 30–50.

[19] SCOTT, K., AND DAVIDSON, J. Low-overhead software dynamic translation. Tech. Rep. CS-2001-18, 2001.

[20] SCOTT, K., AND DAVIDSON, J. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation* (2001).

[21] UNG, D., AND CIFUENTES, C. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM Workshop on Dynamic Optimization Dynamo '00* (2000).

[22] VIEGA, J., BLOCH, J. T., KOHNO, T., AND MCGRAW, G. ITS4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conference* (Dec. 2000), ACM.

[23] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review 27*, 5 (December 1993), 203–216.

[24] WITCHEL, E., AND ROSENBLUM, M. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (May 1996), pp. 68–79.