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

Symantec 1018
IPR of U.S. Pat. No. 8,141,154

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

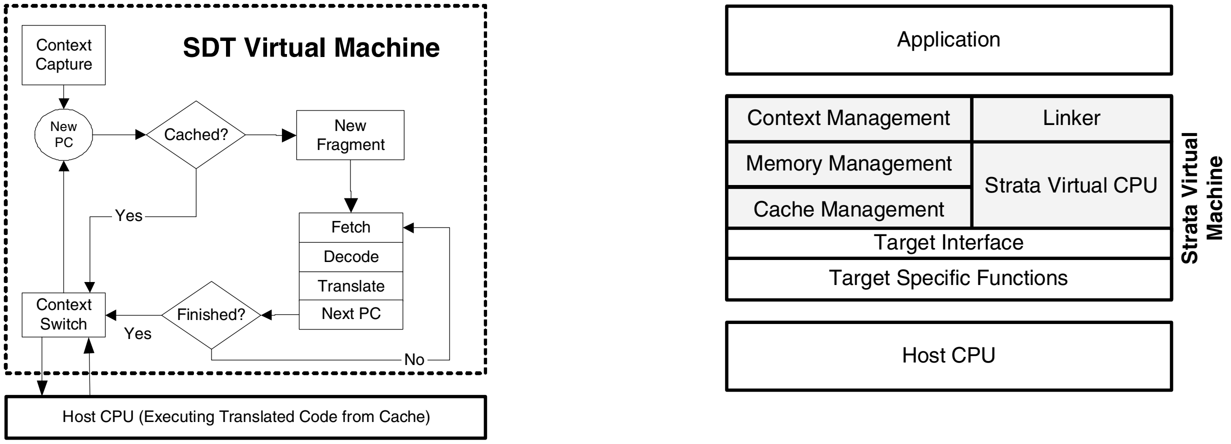
SDT is a technique for dynamically modifying a program as it is being executed. Software dynamic translation has been used in a variety of different areas: binary translation for executing programs on non-native CPUs [6, 7, 21]; fast machine simulation [3, 24]; and recently, dynamic optimization [1]. 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).

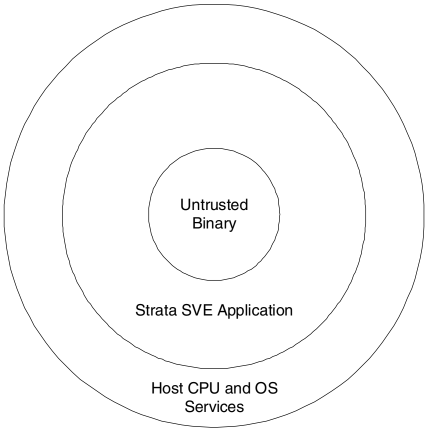


Figure 2: Strata

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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase®
Smarter legal research.