ADAPTIVE SECURITY POLICIES ENFORCED BY SOFTWARE DYNAMIC TRANSLATION

A Thesis
In TCC 402

Presented to

The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment

Of the Requirements for the Degree

Bachelor of Science in Computer Science

By

Paul H. Lamanna

3/25/2002

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in TCC courses.

_____

Approved_____ Technical Advisor_____ Date
Professor Kevin Skadron

Approved_____ TCC Advisor _____ Date
Professor I.H. Townsend

Acknowledgements

A very special thanks to:

Professor Kevin Skadron
Mr. J. Kevin Scott
Professor I.H. Townsend
Professor Dave Evans
Professor Chris Milner
The LAVA Group

Thanks Chris for getting me started and always having an open door.

Table of Contents

Figure List

Glossary

- *Adaptive Policy Enforcement* – a special type of policy enforcement developed in this research that allows a policy to increase or decrease its degree of security awareness in response to pre-defined events.
- *Binary Stream* – the binary code of a running program.
- *Binary Translation* – An extension application to an SDT framework that translates the ISA of programs while they are running so that they may run on hardware other than originally intended.
- *Buffer Overflow* – placing more objects in a buffer than it can hold so that memory is overwritten. This is a common hacking attack.
- *Cache* – a fast memory used for storing frequently used data.
- *Cache Flush* – deleting all data entries of a cache.
- *Code safety* – ensuring that low-level and high-level safety are in place. Low-level code safety guarantees control flow safety memory safety, and stack safety. High-level code safety is specific to the application.
- *Context Switch* – a switch between programs that requires the program data of the first to be saved.
- *Dynamic Policy Enforcement* – enforcing policies while a program is running.
- *Dynamic Optimization* – An extension application to an SDT framework that improves program performance during execution.
- *Executable* – a program that can be executed to perform some task.
- *Exploit* - A technique of breaking into a system, or a tool that implements the technique. An exploit takes advantage of a weakness/vulnerability in a system in order to hack it.
- *Fragment* – a sequence of related code assembled by the fragment builder of Strata.
- *Fragment Builder* – the fetch/decode/translate engine that translates a binary stream into fragments.
- *Fragment Cache* – the cache used to store fragments built by Strata.
- *Generic call* – any function used in a program that is not explicitly used by the operating system, i.e. not a system call.
- *Hacking* – the process by which a malicious user gains unauthorized access into a system, using by exploiting low-level intricacies of a system.
- *Hash Table* – a specific type of data structure used commonly in programming due to its ability to quickly insert and retrieve data entries.
- *Intrusion Detection* – a method of profiling normal system behavior in order to recognize deviations in it which suggests unauthorized access.
- *Patch* – a response to an exploit by security professionals that fixes a specific vulnerability in a system.
- *Format String Attack* - A common vulnerability created by programmers who use tainted input as the format string for `printf()` (a common C function).
- *Denial of Service Attack* - An exploit whose purpose is to deny somebody the use of the service: namely to crash or hang a program or the entire system.
- *Distributed Denial of Service Attack* - A DDoS attack is one that pits many machines against a single victim. An example is the attacks of February 2000 against some of

the biggest websites. Even though these websites have a theoretical bandwidth of a gigabit/second, distributing many agents throughout the Internet flooding them with traffic can bring them down.

- *Security Policy* – a set of rules that manages computer resources such as the file system and memory.
- *Software Dynamic Translation* – Conceptual idea of modifying programs while they are running by inserting a virtual machine between program and processor.
- *SPARC* – Scalable Processor Architecture – one of the architectures Strata is currently ported to, and the platform which all tests were run in our research.
- *Stack Smashing* – a particular hacking attack that uses buffer overflows to circumvent normal program execution by overwriting return addresses on the stack.
- *State Machine* – a method of representing a sequence of actions to be taken by a policy. A state machine consists of a set of an input alphabet, a set of states, and a set of transition relations between the states. Each character of the input is read in sequentially and the machine is traversed.
- *Static Policy Enforcement* – enforcing policies on non-executing code, at compile time.
- *Strata* – an implementation of the concept of SDT currently under development by Kevin Scott and Professor Jack Davidson at the University of Virginia Laboratory for Computer Architecture.
- *System call* – a function used by the operating system used to control resources. They are often exploited in hacking attacks, so they are the focus of a number of policy enforcement schemes including ours.
- *Target Code* – the source code on which a security policy is applied.
- *Trampoline* – a special function used to retain control of a process in Strata.
- *Translation* – the process by which Strata mediates program execution, specifically builds fragments for subsequent execution.
- *Transparent Execution* – when a program runs without need of user assistance or making any changes to application semantics, so that the user does not necessarily know that it is even running.
- *Un-trusted Binary* – code from an unknown source that
- *Virtual Machine*- any multi-user shared-resource operating system that gives each user the appearance of having sole control of all the resources of the system.
- *Working Set* – the 10% portion of a program that runs 90% of the execution time.

Abstract

This report describes a software system that transparently supports the enforcement of user-specified security policies. The system was implemented as an extension to a Software Dynamic Translation (SDT) framework known as Strata developed at the Laboratory for Computer Architecture at the University of Virginia (LAVA) by PhD student J. Kevin Scott and Professor Jack Davidson. SDT technology is an exciting new innovation in computer science that has the ability to monitor and modify programs while they are running. The policy enforcement scheme presented here provides a simple Security API available for policy authors to implement a wide range of security policies in the C programming language. I have contributed to this application by extending both the API and the enforcement mechanism to enforce a special type of policy defined in this report as *adaptive* security policies. While these policies do not claim to act using their own intelligence, they do have the ability to alter policy rules based on user-specified responses to user-specified events.

It is our conclusion that the system we have presented is beneficial to the security community. SDT technology allows enormous flexibility in policy specification, as does the nature of our API which is supported in the widely used C programming language. Policies can be applied to programs without the need of source code manipulation prior to execution, which aids in the degree of execution transparency. This capability alone transcends the options presented by static policy enforcement. Our system already performs a derivative of execution monitoring without the need of operating system assistance, and unlike many static solutions, we are not limited to the enforcement of a single policy. The dynamic information provided at run-time allows dynamic and adaptive policies to be written that at least equal, if not supercede, the scope of any static policy and require significantly less programmer effort. Our enforcement mechanism, Strata, makes all of this possible.

The enforcement of adaptive policies only increases policy flexibility by giving policy authors the ability to increase or decrease the degree of security a policy applies during execution. By responding to pre-defined events, an adaptive policy can justify threats before enforcing computationally expensive policies. While this savings in performance has not been fully quantified due to the wide range of policies our system may potentially support, by simply providing new options to security engineers the probability of attaining security solutions increases.

Chapter One: Introduction

This report describes a software system that transparently supports the enforcement of user-specified security policies. The system was implemented as an extension to a *Software Dynamic Translation* framework known as *Strata*, which is an exciting new innovation in computer science that has the ability to monitor and modify programs while they are running. The policy enforcement scheme presented here provides a simple Security API available for policy authors to implement a wide range of *security policies* in the C programming language. I have contributed to this application by extending both the API and the enforcement mechanism to enforce a special type of policy defined in this report as *adaptive* security policies. While these policies do not claim to act using their own intelligence, they do have the ability to alter policy rules based on user-specified responses to user-specified events.

Computer security is becoming an increasingly important issue in this age of global communication, electronic commerce, and the Internet. It is highly researched for several reasons, its relevance to consumer privacy and due to the persistence and resourcefulness of attackers. Hackers are constantly discovering vulnerabilities in systems leaving behind a legacy of malicious programs known as exploits. As long as hackers have been hard at work causing problems, there have been efforts to prevent the damage they cause. These efforts have been both preventative of attacks and responsive to them, statically and dynamically. One such application, developed at the University of Virginia by Professor Dave Evans and Dave Larochelle, statically analyzes C programs for a certain type of *exploit* [19]. This would be an example of a preventative measure, but also a very static solution, since the source code is required before compilation and the ques-

tionable code is never run. A similar task is also accomplished dynamically (a dynamically linked library) at run-time by a program developed at Avalya Labs named Libsafe [18, 16]. This application is also preventative, and accomplishing the same end goal, but doing so in a different manner. Another portion of the these efforts have evolved into the field known as *intrusion detection* systems (IDS), which as the name suggests, strives to profile the behavior of normal systems so that attacks may be detected while taking place. This approach is drastically different from the earlier examples and is a good indication of the diversity of computer security applications. Consider yet another method, the idea of enforcing user-defined security polices. This approach offers a great deal of flexibility because policies can be tailored to provide different levels of security for different platforms. Policies are most commonly defined as a set of safety properties that place constraints resource operations [6]. By controlling computing resources and being suspicious of how processes use them, a policy may prevent attacks by reducing the mobility of malicious code.

Different users have a need for differing security applications, with different levels of security sensitivity. Certainly, the computer network in the Department of Defense building would require a greater number of security measures than a personal webpage. On a general level though, users value their privacy and security but are extremely sensitive to a decline in the performance of their machines. All security applications produce some performance overhead, but in order to achieve widespread use they must operate with some degree of *transparency*, to provide the maximum protection possible with the least amount of user interaction required and performance penalties incurred. The dy-

namic capabilities of SDT certainly aid this degree of transparency and offer new approaches for current security practices.

Software Dynamic Translation (SDT) technology is a relatively recent and exciting innovation in computer science. SDT systems have the ability to monitor and/or modify the native binary execution stream of a program while it is running, and they can do so transparently without the need of pre-compilation or programmer assistance. Implemented entirely in software, SDT systems run between a process and its host CPU, interrupting the execution stream immediately before it reaches the processor, and achieve a capability to control execution dynamically. It is important to emphasize that a run-time perspective of an executing process has unique and exciting benefits. Various implementations of SDT have surfaced over the past few years, realizing concepts such as *binary translation* and dynamic optimization. Binary translation technology allows for the portability of programs between differing architectures. By translating a *binary stream* from one instruction set architecture to another, a program can run on hardware other than what was originally intended. *Dynamic optimization* technology actually improves the performance of programs while they are running, in contrast to a compiler which performs its optimizations statically at compile time. Perhaps what is so impressive about these technologies is that they accomplish all of this without ever having to manipulate the program at any time prior to execution. This advantage is not only valuable to the portability and flexibility of these applications but the catalyst of this thesis project. Due to this unique capability, SDT technology has an enormous potential for success in the field of computer security.

I have contributed in the implementation of an SDT application for enforcing se-

curity policies like those described previously, only the policies have the ability to adapt to changing security requirements during the execution of one or more processes. Such an application has the ability to turn on or off its security monitoring or increase or decrease its level sensitivity based on the current state of the system. It is this adaptive capability that I have contributed to the application. The application:

- *supports the concept of execution transparency,*
- *supports the flexibility and portability of the policies by enforcing them in a platform independent manner,*
- *maximizes the scope of security policies by making them adaptive to changing security requirements, and*
- *minimizes performance overhead associated with enforcing the security policies.*

A major motivation for having adaptable security policies is that enforcing security policies is inherently computationally expensive. An expensive policy should only be enforced when it is considered necessary, and SDT technology makes this transition possible. Recent research in policy enforcement describes applications based in the operating system kernel, since system calls are readily accessible from this point [5]. SDT allows such system call access in addition to the rest of the execution stream, so policy enforcement with SDT does not need the aid of the operating system. Separating this functionality from the OS is ideal, because policies involving more than system call monitoring would be difficult or impossible to enforce by the kernel. Simply put, designing a kernel policy enforcement infrastructure fixes the type of policies that are enforceable. As a result, policy enforcement could complicate the kernel implementation which could introduce bugs, make maintenance more complicated, and even decrease performance.

Any advance made in any area of computer security produces numerous impacts. These impacts affect not only the common user but also global networks of shared information, corporations, and even the hackers who invoke the threat. When security methods are improved, users gain an improved sense of security. This new environment nurtures progress. Whether the field is communication, commerce, or research, increased security allows professionals to concentrate on their work rather than worrying about its exploitation. It is important, however, that users do not gain a false sense of security, by overestimating the capabilities of security products. Such a scenario produces negative effects caused by users who place themselves in vulnerable situations they otherwise would not consider. Therefore, it is imperative that the scope of security software is well documented and tested, so that users are fully aware of its protective abilities and limitations. Successful security systems fulfill four general requirements. They are 1) well documented, 2) lightweight (satisfactory performance on a given system), 3) easy to set up and maintain, and 4) and successful at defending a system from all of the attacks advertised. Notice that these requirements are associated not only with the level of protection provided by the product but with how the product affects the performance of the system. Recall that SDT systems have the capability to fulfill these requirements, resulting in a high degree of success.

The average computer user has little idea of the risks involved in everyday use. And even the more informed user would not be able to detect that an attack is taking place that simply violates privacy and causes no noticeable harm. An application such as the one proposed offers the common user a simple and transparent mechanism to deal with security issues so they do not have to. Equally, the major attraction of this tool for

the security engineer is its flexibility for implementing an extensive set of security policies. In addition, the policies themselves have an increased individual ability and scope of functionality due their adaptive ability. This introduces a wide range of possibilities for security engineers to combat the never-ending challenge of computer security. Security engineers will, in contrast to the common user, have a detailed knowledge of the system and the current state of security. With the dynamic content made available by this tool, they will be able to construct a wide variety of security policies specifically tailored to the system, the attacks in question, and to reduce performance overhead. While the common user also has this ability, this larger system offers more opportunity for diverse application due to a much greater number of users and a higher probability of security breaches. It is in this forum of research that the tool will be thoroughly tested and information and policies themselves may be passed down to the common user. Since the proposed policy enforcement scheme is implemented outside of the operating system it aids to the flexibility and portability of the tool so that this may be possible. This type of testing is invaluable because the resourcefulness of hackers cannot be underestimated. It is important to understand that new security applications affect the hacker just as much as they provide a sense of security to those who use them. Therefore, new security applications are usually the most susceptible to attack, and must be made as invisible to the hacker as possible to lessen any negative effects of its deployment.

The following chapters in the report attempt to give sufficient background in computer security and Software Dynamic Translation to understand the motivation for this project and the methods by which it was accomplished. Chapter 2 is a review of relevant literature. Here a general background is provided in all aspects of computer se-

curity and SDT applications. Chapter 3 discusses the definition and enforcement of various types of security policies, including the adaptive type developed during the research for this thesis. Here our approach is contrasted with applications developed in other security research, which mostly because we use SDT to enforce our policies. Chapter 4 discusses the operation of our SDT framework, Strata, in detail. With the information provided in Chapters 3 and 4, Chapter 5 presents the modifications made to the Strata framework and security API to facilitate the enforcement of adaptive policies. Chapter 6 gives a tutorial for writing effective adaptive policies and several examples. Finally the report is concluded in Chapter 7.

Chapter Two: Literature Review

This chapter attempts to provide a background in computer security, policy enforcement, and Software Dynamic Translation through discussion of related literature. This background is sufficient enough to understand the motivations for this project.

This thesis project draws literature from several differing areas of study. A complete understanding of SDT is required to implement a new application for it. This knowledge not only includes SDT system design, but a great deal of underlying computer architectural principles. It is important to comprehend how a computer works from a low enough level of abstraction to be equal with the perspective of an SDT system (the assembly language level), and depending on which architecture is present this operation will differ from machine to machine. An understanding of machine level operation ties in intimately with computer security. Since the majority of security threats evolve from exploiting the low-level intricacies of a system, architectural knowledge is essential in undertaking security research. This practice of exploitation is known as *hacking*, and it is usually accomplished by any means necessary. The field of Computer Security attempts to discover, prevent, and respond to hacking. Like any other field, its goals are accomplished in various manners and through various abstractions, from low-level code certification to user and program behavior profiling to policy enforcement. All of these methods have proved to be effective to some degree in practice, but the resourcefulness of attackers presents a seemingly endless challenge for security engineers. The literature presented in the following sections will provide sufficient background, in SDT, computer architecture, and computer security, to enable the reader to fully comprehend the motivations behind this research and the call to arms for future research. With this foundation in place, an account of the additional concepts related directly to this project can be clearly

presented.

An implementation of SDT, named *Strata*, is currently under construction at the University of Virginia by the Computer Science PhD student James Kevin Scott. The Strata infrastructure provides a framework for implementing new SDT applications, so Strata has the ability to support functionality such as binary translation and dynamic optimization [4]. Hewlett Packard's Dynamo [1] is a transparent dynamic optimization tool which provides a similar framework, only code is also integrated to optimize PA-RISC binaries. Several other examples of dynamic optimization technology are Vulcan (IA-32), Mojo (IA-32), and DBT (PA-RISC). IBM's Daisy is an example of binary translation technology, translating the VLIW instruction set to PowerPC (Daisy also performs some dynamic optimization) [2]. FX!32 (IA-32 to Alpha), UQDBT (IA-32 to SPARC), and Transmeta's Code technology (IA-32 to VLIW), are other examples of binary translation technology. These products display the commercial success of SDT technology as well its flexibility to implement a wide range of services across multiple platforms with a single tool.

Both of the technical reports on Strata, entitled *Strata: A Software Dynamic Translation Infrastructure* and *Low-Overhead Software Dynamic Translation* give an excellent depiction of how Strata arbitrates program execution. Strata gains control of a process and maintains this control through special context switches, called trampolines, between its framework the running process. Each executed instruction is dynamically translated to become part of a cached sequence of code known as a *fragment* [4]. When the code is finally executed on the host processor it is done so from the fragment cache, a

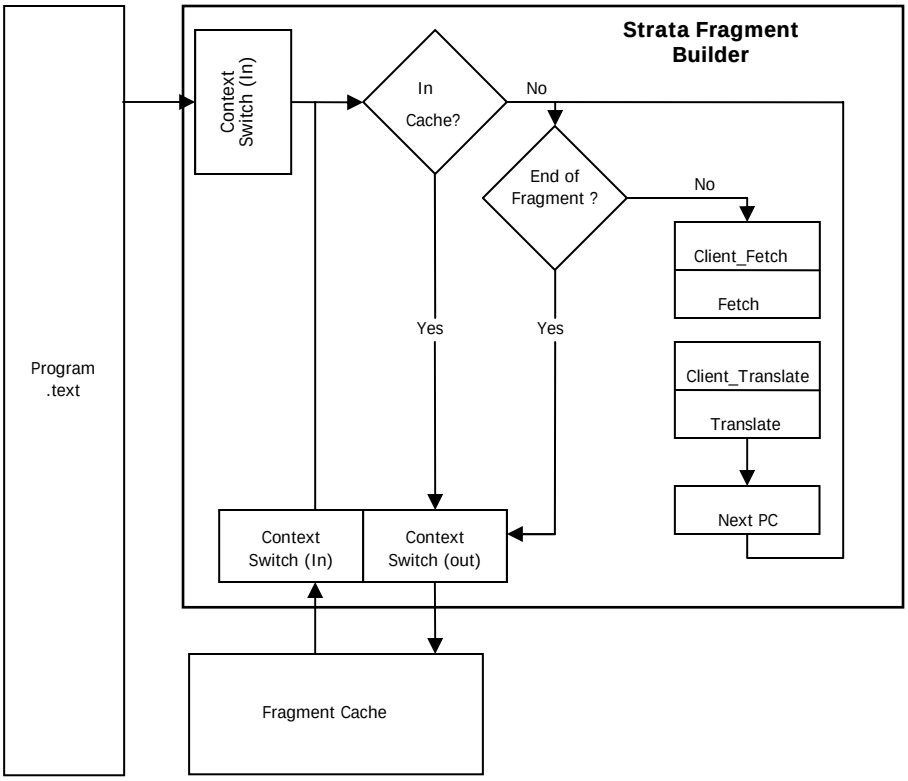


Figure 1. Strata Translation Process [4].

very fast memory resource containing the newly translated instruction. Note that this functionality provides a facility to modify this code before it is placed into the fragment cache, and before actual execution on the processor. The modification may be, and certainly not limited to, a code optimization or an insertion of security checking code. Figure 1. illustrates a portion of the architecture of Strata, specifically the formation and caching of fragments, which is known as the fragment builder.

In order to define this thesis, I had to determine what modifications Strata should make to a program to make its operation more secure in some way. This task required sufficient background in the concepts of *hacking* and computer security. Generally speaking, hacking tends to exploit the low level intricacies of an operating system and/or

architecture such as register organization, memory access and function calling and returning. A common, and maybe obsolete, example of hacking is a *buffer overflow* exploit or a stacking smashing attempt. These exploits take control of a process and force it to execute malicious code [15]. Denial of service (DoS) attacks or distributed denial of service (DDoS) attacks contrastingly render a machine useless by bombarding its resources and overloading the CPU. The list goes on to include brute force attacks, IP spoofing, format string attacks, session hijacking, Trojan horses and many more. What is done to try and prevent these attacks, or to try and discover that they have happened and to take appropriate action, is the basis of computer security.

Various areas of computer security have evolved with these goals in mind. Each area seems to approach the task at hand through its own level of abstraction. Perhaps the lowest level of abstraction is pursued by a concept known as *code safety*. Code safety strives to provide control flow safety, memory safety, and stack safety to ensure programs behave properly [21]. Low-level code safety supports higher level security mechanisms, especially those that allow code modified with security checks to run. Without this type of safety in place a hacker could get around the security checks inserted to ensure that the code is secure. Software Fault Isolation [20] and Proof-Carrying Code [22] are two examples of mechanisms to provide low-level code safety. For any security extension to Strata to be successful in reality, low-level code safety must be ensured so that Strata's actual operation cannot be circumvented via hacking techniques. This task, however, is beyond the scope of this project, and will be left for future research.

One approach to dealing with the problems caused by hackers is to face the problem head on, constructing software that prevents specific types of attacks. LCLint stati-

cally analyzes C programs for buffer overflow exploits [19]. Libsafe is a dynamically linked library that performs bounds checking on certain C functions to prevent stack-smashing at run-time [18,16]. Strata, actually, has the ability to completely prevent stack-smashing attempts with the addition of only several lines of code. This functionality along with a very simple policy enforcement mechanism is described in the technical report entitled *Low-Overhead Software Dynamic Translation*. Other efforts attempt to take a more general approach to the problem in the hopes of finding a more flexible solution to the continuously evolving hacking arsenal. Intrusion Detection Systems (IDS) is one such effort which, as the name suggests attempts to determine accurately when a system is under attack so that it may take proper action. It does this by profiling normal system behavior through system calls, user actions, and other past behavior. Any departure then from this normal behavior could be considered an attack [12]. A drawback of this approach though is that behavior profiles can never be one hundred percent accurate, meaning that often, innocent processes get treated as malicious ones. Yet another area tries to combat attacks by managing system resources through the use of security policies. Naccio is one such architecture which allows for the expression of security policies in a platform independent manner. These policies are then enforced by an application transformer, which rewrites the entire *executable* including wrappers around code that needs to be monitored by the policy [6]. There is research also in how to efficiently construct and place these wrappers, which are simple *state machines* that sit idle and listen for an event to occur. Upon this event the state machine takes action as defined by the policy [5]. The transformation invoked by Naccio is conceptually similar to how Strata translates executing code and caches it for subsequent execution. Policy enforcement in Strata

is similar also, only this policy has the ability to change based on the current state of the system. The adaptive policy developed in this report is based on a hybrid of current policies, such as the Chinese wall policy, audit policies, and information flow policies [8], and utilizes the ability to completely switch between the policies as the situation permits. Strata provides a basic security API that supports the incorporation of this type of policy enforcement into its infrastructure, supplying policy authors with functionality to easily watch *system call* and *generic call* instances [11].

Chapter Three: Security Policies

This chapter introduces security policy enforcement and specification techniques used in some current security systems. A distinction is made between static and dynamic policies and how they are enforced. Finally, adaptive policies are introduced, which are specific to this report.

Introduction

Security policies monitor how processes manipulate system resources and use the information gathered, known as state, to determine if a security violation is about to take place. A policy is a set of rules that specifies how resources should be used. In order to ensure that a process follows these rules there must be some mechanism in place to provide a means for both policy specification and policy enforcement. Using policy rules as inputs, the mechanism must watch for events that would break the rules, and keep track of the state of the program by remembering what happened when the watched events actually occurred. Generally, specification mechanisms are implemented using a specification language defined by the security system creator. The Naccio system [6] uses its own language consisting of a set of resource manipulations and additional user-specified security checking code. Some systems employ the use of finite state machines [7] to define their policies. Further compilation then converts these into recognizable syntax. Our SDT application greatly simplifies the specification process by providing a security API that allows policy authors to write policies in the C programming language while providing them with additional security functionality.

Following specification, enforcement mechanisms apply the policies to *target code* from one of two possible vantage points. The first is to do so from a separate ad-

dress space (the enforcement code resides in its own memory space separate from the target code), which requires a transfer of control into the enforcement code each time a watched event occurs. When a resource operation takes place that has been deemed questionable by the security policy, a *context switch* occurs to allow the policy code to execute. Such an enforcement approach is computationally expensive due to the context switches. It also limits the range of events that can be watched, which degrades the expressive potential of the policy itself. This side effect occurs simply because some events are easier to watch than others. Typically low-level (machine level) events that are application independent such as the execution of a system call are much easier to watch than higher-level (application level) application dependent events. System calls have traps that can be easily watched whereas the use of a macro in a word processing program would be considerably more difficult to recognize. Complexity is added to the enforcement mechanism when it must determine ways to watch for events that are not explicitly defined by particular low-level events. The advantage of using a separate address space is that the enforcement mechanism itself is protected from subversion.

The other possibility for enforcing policies is modifying the source code of the program to include the policy code, inserting security checks around questionable code to achieve the event watching process. The policy code is essentially merged with the program code. They run as a single program from the same address space, saving the cost of context switches. SASI inserts its state machine information after each instruction in the target code (prior to execution), so that during execution the machine is traversed (note that the state machine is the specified policy and the enforcement mechanism combined). Naccio rewrites the entire executable to include wrappers functions, which when exe-

cuted call the policy code. This type of enforcement, which attempts to merge policy code with the target code, is conceptually similar to our scheme, only ours allows for the policy code to be continuously re-merged with the target code while it is running. Using SDT technology to enforce our policies, which will be explained in the following chapter, made this capability possible. The following sections of this chapter distinguish between several general types of policies used in the evaluation of our enforcement scheme, keeping in perspective how each type is enforced.

Static policies

A static policy system is generally one in which a limited set of policies can be enforced on a system because the policies are applied to programs at compile time, which is prior to execution. Some systems perform source code analysis to determine if code is likely to produce security violations (or execution errors in the simplest case). LCLint [19] analyzes non-executing source code of C programs in an attempt to determine if buffer overflows could occur during execution. This is a perfect example of a static policy, both because the enforcement of the policy takes place at compile time and because there is only one policy being enforced. By restricting analysis to only the C language also adds to the static nature of the system. There is a whole class of policy schemes targeted at spotting programmer errors that may not be visible to a compiler, but often lead to security breaches. A common example of this is misusing the `printf` statement in C to cause what is known as a Format-string vulnerability. A programmer could either do this by mistake or on purpose with malicious intentions. Hackers use these and other vulnerabilities intentionally to break into systems.

Static policy enforcement is limited in both the range of policies it supports and its dependence on specific languages. *Dynamic policy enforcement* attempts to alleviate these limitations by monitoring programs while they are running.

Dynamic policies

Dynamic policies can use run-time information to make decisions about the state of the system. Some dynamic enforcement schemes, like the static ones, only enforce one policy in one language but they do so at run-time when a greater amount of information is available to them. Libverify, Libsafe, Libformat, and StackGuard [16,18] all attempt to prevent buffer overflows and stack-smashing this way. Others are much more flexible and allow a number of policies to be enforced dynamically through a technique known as execution monitoring [5,17]. The SASI system is one example. Janus [5,17] uses execution monitoring functionality provided by the operating system to observe the use of system calls, which it evaluates through the use of a specified policy. Note that these systems must also provide a means for policy specification. In our case, SDT technology makes run-time information available to dynamic policies by merging policy code with the target application during the translation process. All of our policies are inherently dynamic, because they are actually applied at run-time. Like Janus, we also monitor system calls (among other things); only it is done without the need of operating system assistance. In contrast to SASI, our specification mechanism is greatly simplified.

While other dynamic systems vary in capability, our enforcement scheme ensures access to function parameter values, memory locations, and pointer values allowing our policies to make more informed decisions than static policies about the security state of a

running program. Such access also greatly eases the burden on the policy author, making the policy code generally more refined, robust, and compact.

Adaptive policies

An adaptive policy is one specific to our enforcement implementation. It is a special type of dynamic policy that allows policies to change their own rules during execution. The primary motivation for providing this functionality is to give the policy author the option of varying the level of security sensitivity a policy exhibits. Higher levels of sensitivity require watching a greater numbers of events, incurring higher translation and computing costs. An adaptive policy attempts to only enforce expensive portions when a security threat is fully justified. This bestows additional responsibility on the policy author to know enough about computer security to determine what occurrences justify changes to the policy. For instance, consider a program manages to set the user-id to root, placing the process in super-user mode and giving it free reign over system operations. In a case such as this, enforcing a more expensive policy that has more of a chance of preventing malicious acts is certainly in order. An adaptive policy can respond by invoking this increase in security awareness. In order to do so, the policy author first must know that setting the user-id to root is a pre-requisite for certain security breaches. The adaptive policy then watches for this specific event, and when it occurs, the policy rules change to enforce a new policy specifically tailored to thwart super-user exploits. The key here is that the new policy is enforced only when it is necessary to do so, when the threat of a security violation is great.

It is important to understand that these policies do not exhibit any form of artificial intelligence. All actions and responses must be defined in the policies. In the previous case, the policy must be defined to watch the `setuid(0)` event, and the response to this event must also be defined to switch to the other user-specified policy. Currently, using system calls, and generic calls, a policy writer must make decisions on what calls to watch based on expected security threats. Decisions must be made on the priorities of events that cause changes in the policy (the adaptive behavior), so that security sensitivity may be increased or decreased.

Chapter Four: Operation of Strata

This chapter explains how the Software Dynamic Framework, Strata, mediates program execution through a process called fragment translation. Other topics include fragment optimizations and how Strata takes control of a process and retains it.

Introduction

Software Dynamic Translation (SDT) technology has the ability to monitor and modify a program while it is running. Because the program suffers from no visible changes semantically, SDT is said to operate transparently. By introducing an additional layer of software between program and processor, SDT systems create a *virtual machine* that mediates the binary stream of a program and thus controls what is actually being executed on the processor. This ability is a foundation for building applications that require this control. The SDT system used in this project is called *Strata*, and it is an implementation of the concept of Software Dynamic Translation. It is currently under development at the Laboratory for Computer Architecture at the University of Virginia (LAVA) by PhD student J. Kevin Scott and Professor Jack Davidson. Strata provides a extensible framework, or interface, to implement SDT applications. One such application extended Strata to enforce user-specified security policies. This project added functionality to that application. It is the focus of this chapter to explain the operation of the actual framework of Strata and how it accomplishes dynamic translation. The security extension application will be introduced in the following chapter.

Strata is composed of approximately 8000 lines of C code, and requires no hardware support other than access to the cache. Strata currently runs on SPARC/Solaris and

MIPS/Irix platforms, the first of which was used in this project. Portability to each architecture requires about 30% of the 8000 lines of source code.

The following Figure 2 illustrates the architecture of Strata. Using the basic services shown in this figure Strata controls and mediates the execution of a program binary. These services include memory management, *fragment cache* management, application context management, a dynamic linker, and a fetch/decode/translate engine. The fetch/decode/translate engine is also known as the fragment builder. Figure 2(a) shows the *fragment builder* and Figure 2(b) shows the abstract architecture as it relates to the application it is translating and the processor.

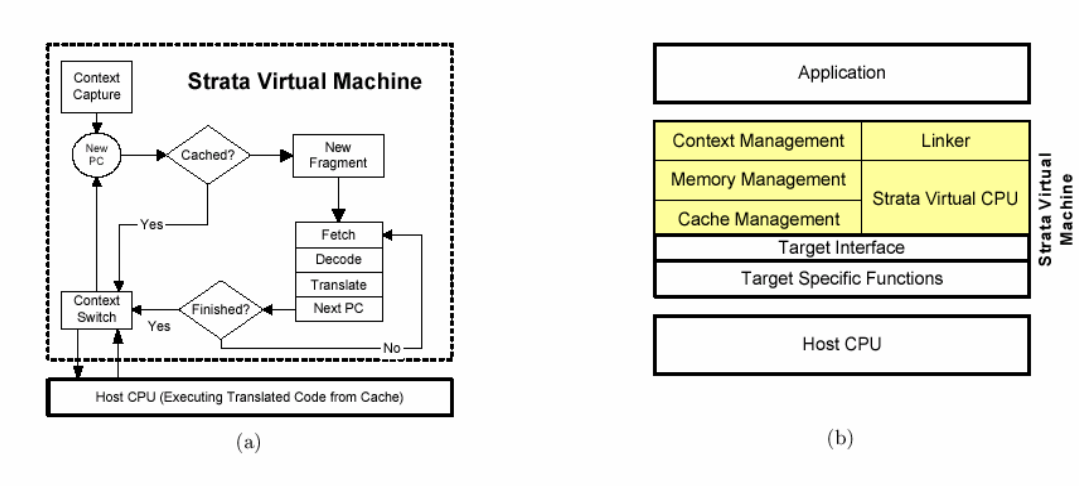


Figure 2. Architecture of Strata [11].

Utilizing a very fast memory (cache) as storage, a representation of the *working set* of a program is sampled from the binary stream for later execution. This mediation of the binary stream is accomplished through a process called *fragment translation*, which builds the actual code fragments that make up the working set. These fragments are then placed into the *fragment cache* for ensuing execution. Applications that are built to run in a SDT system exploit the translation process to achieve their goals. SDT gives

its applications a run-time perspective of a process, which means that the majority of program data is current and available for manipulation. This data is used to make decisions about what code to actually run on the processor. Binary translation, an extension application of SDT, allows a program to execute on foreign hardware by transforming its Instruction Set Architecture (ISA) on the fly. The current instruction that needs to be executed and the run-time data that is used by the instruction are all available to the translator. With this information a decision can be made about the corresponding instruction on the foreign hardware that will accomplish the same task as the original instruction. The SDT framework simply makes this information available to make the process possible. Dynamic optimization applications analyze code fragments for optimization possibilities. Code fragments considered to be improvable are modified and then cached so that each time they are executed the process experiences a performance gain. Over time, the entire working set of the program materializes in the fragment cache, requiring less and less work to be done by the SDT application. Since the code is optimized, the program is running faster (or more efficiently) than it would have originally.

Building a representation of the working set of the program is the basic function of the translation engine. Statistics show that the working set of a program usually consists of approximately ten percent of the source code of a program, and this code runs approximately ninety percent of the execution time. This fact allows the SDT framework to make up for the overhead introduced by its translation process. As the working set is built, it is up to the specific application to decide what to do with the code. Optimization programs simply try to make this code run faster, while binary translation programs perform ISA translation. The application developed in this project actually monitors the

code for security violations. It also inserts instructions into the code fragments that cause questionable code to be rerouted into user-defined security policies when it is executed on the processor. These policies then perform security checks on the state of the program according to the rules set by the policy author. The SDT application gives the policies a run-time perspective of the state of the program. The following Figure 3 shows this basic process of merging policy code with target code.

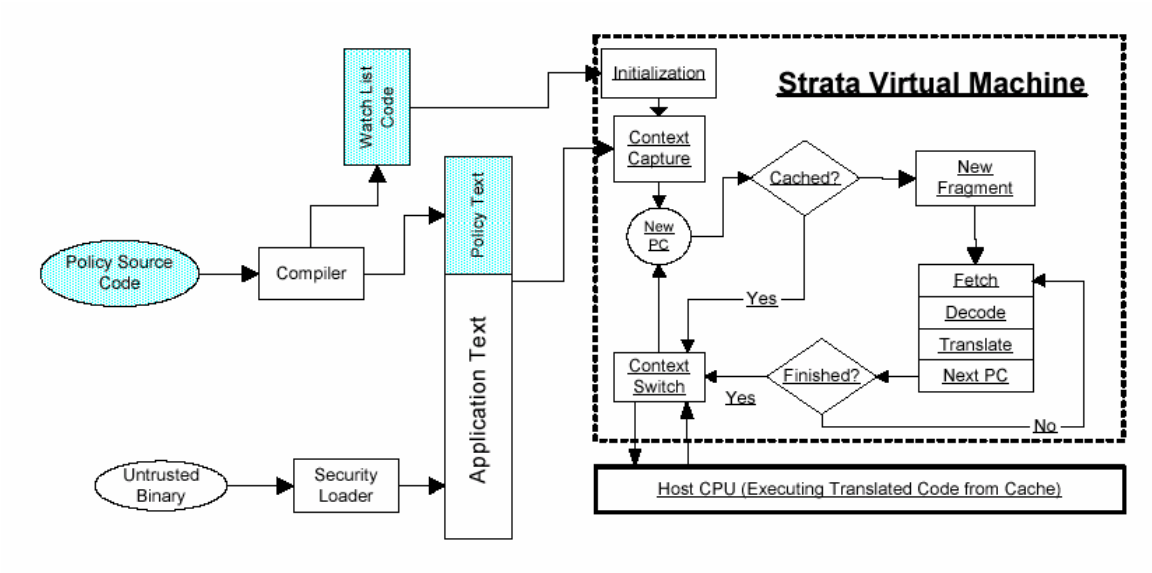


Figure 3. Merging Policy and Target Code

What makes SDT so useful and impressive is that it can function without ever having to see the programs it translates prior to execution. This ability is a major motivation for computer security applications, where the code that is being run is mobile and generally not trusted (e.g. attached e-mail executables). An important part of the operation of Strata is how it takes control of these programs and how it keeps control throughout translation and execution. Ensuring this control, translation and fragment creation can take place. Built in services of the Strata virtual machine also perform code optimizations on the fragments in an attempt to more quickly overcome the overhead of the

translation process. Translation allows the particular extension application to Strata that is enabled to analyze and/or modify the code. Strata then finally inserts the fragments into the cache for subsequent execution and continuously manages the cache while further translation takes place. The following sections discuss how Strata retains control over a process and some of the basic optimizations it performs on fragments.

Strata Control Mechanism

In order for Strata to monitor a process, it must take control of it. It does so using functions `strata_init`, `strata_start`, and `strata_stop`. These functions must be placed in the main function of a program in order for Strata to take control. In the future these functions will not be necessary, rather Strata will only need to be given the compiled executable binary (thus giving it the ability to run transparently). The following figure 4 shows how these functions are used to take control of a process (in the C language). Note that `strata_init` must be called after variable definitions in `main`.

```
 1  int main (int argc, char* argv[]) {
 2          .
 3          .
 4          .
 5          // Local Variable Definitions
 6          .
 7          .
 8          .
 9          strata_init();
10          strata_start();
11          .
12          .
13          .
14          // Target Code or call to Target Code
15          .
16          .
17          .
18          strata_stop();
19          return 0;
20  }
```

Figure 4. Strata takes control of a process.

Strata uses functions called *trampolines* to retain control over a process. It must use them at the end of fragments, which are usually branches, returns, or calls. A trampoline returns control back to Strata's fragment builder after the end of a cached fragment is reached, so that translation can continue.

Fragment Optimizations

Strata performs some optimizations on the fragments during translation to reduce overhead introduced by the translation process. Most of this overhead is due to a high number of context switches, which incurs the expense of saving program data to memory. These optimizations [4] include:

- **Partial inlining**
  This technique is a code optimization which replaces function calls with the actual code of the function. Instead of coping the entire function body, only code segments that execute frequently, called hot regions, are rewritten. The remainder of the function, known as a cold region, is then called after the copied portion.

- **Fragment linking**
  After every fragment executes a context switch occurs to go back into the fragment builder. From there the builder determines if the next fragment is cached or if it needs to build another one. Fragment linking attempts to reduce the number of context switches by linking fragments together as their order is determined. Linking is accomplished by rewriting the trampoline at the end of a fragment to point to the beginning of another.

- **Efficient indirect branch handling**
  This technique also aims to reduce the number of context switches that take in the execution of a program, by using a small cache to store how branch-target addresses map to their fragment cache locations. The instructions necessary to look up this information are translated into the fragment itself in order to avoid the context switch. The cache is direct-mapped.

Chapter Five: Attempting Security with Strata

This chapter presents our system for writing dynamic and adaptive security policies to be enforced by Strata. This includes a general tutorial in policy writing as well as a discussion of the modifications made to the Strata security extension application to support the enforcement of our adaptive policies. Using the functionality provided in the Security API we present a number of examples of flexible dynamic and adaptive security policies. Finally, we present an evaluation of the flexibility and practicality of the Security API.

Introduction

Under the direction of Professor David Evans, we have attempted to write a set of policies that illustrate useful security functionality. While we do not commit that these policies are accurately representative of those used in real world systems, we do believe that they provide reasonable evidence of our system's potential for future success. Most of our policies involve file manipulation and auditing (information logging).

Security API

The Security API provides a means for security policy specification. It is currently supported in the C programming language, and is intended to be language independent in future releases. Recall that a security policy specifies how the computing resources in a system may be used. Since most security vulnerabilities arise from resource misuse, policy enforcement provides an effective defense against the various attacks of hackers. Computing resources are managed by the operating system through the use of operating system calls, so policies can gain an idea of how resources are being used by monitoring system calls. The functionality provided by the API gives policy authors

simple and convenient access to operating system calls and generic function calls. This allows them to specify which calls to monitor during the execution of an *un-trusted binary*. The API consists of the following nine functions.

```
void init_syscall();
watch_syscall(unsigned num, void *callback);
unwatch_syscall(unsigned num)
watch_call(void *func, void *callback);
unwatch_call(void *func);
void strata_callback_begin (void *);
void strata_callback_end (void *);
void strata_syscallback_begin (unsigned);
void strata_syscallback_end (unsigned);
```

This simple API facilitates system and generic call monitoring, allowing user-specified policy code to execute when the monitored calls are made. The following section will illustrate how these functions can be used security policies to aid in resource misuse prevention.

Writing Policies

A policy is simply a C program with a specific structure. Consider a policy to prevent opening the file `/etc/passwd`. Figure 5 illustrates the intended program structure of this policy.

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <strata.h>
 4  #include <sys/syscall.h>
 5
 6  int myopen (const char *path, int oflag) {
 7          int fd;
 8
 9          strata_syscallback_begin(SYS_open);
10
11          makepath_absolute(absfilename,path,1024);
12          printf("\n%s\n", absfilename);
13
14          if (strcmp(absfilename,"/etc/passwd") == 0) {
15                  strata_fatal("Request Not Allowed!");
16          }
17          fd = syscall(SYS_open, path, oflag);
18
19          strata_syscallback_end(SYS_open);
20
21          return fd;
22  }
23
24  void init_syscall() {
25          (*TI.watch_syscall)(SYS_open, myopen);
26  }
```

Figure 5. Intended Basic Policy Structure.

Current progress in the Strata framework requires that the policy to take a slightly different structure. The policy code must exist in the same executable as the target code it is applied to. The target code is must be called, or written, in the `main()` function of the policy code. Using `strata_init()`, `strata_start()` and `strata_stop()`, control of the target code must be transferred to Strata manually. The dynamic linking of the `init_syscall()` function is currently not functional so it also must be manually called

in the main function before the target code executes. Figure 6 shows the same file-open policy adhering to the current policy structure requirements.

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <strata.h>
 4  #include <sys/syscall.h>
 5
 6  int myopen (const char *path, int oflag) {
 7          int fd;
 8
 9          strata_syscallback_begin(SYS_open);
10
11          makepath_absolute(absfilename,path,1024);
12          printf("\n%s\n", absfilename);
13
14          if (strcmp(absfilename,"/etc/passwd") == 0) {
15                  strata_fatal("Request Not Allowed!");
16          }
17          fd = syscall(SYS_open, path, oflag);
18
19          strata_syscallback_end(SYS_open);
20
21          return fd;
22  }
23
24  void init_syscall() {
25          (*TI.watch_syscall)(SYS_open, myopen);
26  }
27
28  int main(int argc, char *argv[]) {
29       .
30       // Local Variable Definitions
31       .
32       strata_init();
33        init_syscall();
34        strata_start();
35       .
36       // Target Code or call to target code.
37       .
38        strata_stop();
39        return 0;
40  }
```

Figure 6. Current Basic Policy Structure.

This structure was used in all of our test policies, and though some semantics may differ in the future, the basic policy structure will remain consistent. In general, a policy consists of a series of callback functions for system calls and generic calls. A callback function is simply a substitute function that executes policy code in place of a watched call. The policy author uses the `watch_syscall(unsigned num, void *callback)` and

`watch_call(void *func, void *callback)` functions in the policy code to tell the fragment builder which calls to watch and reroute during translation. The fragment builder replaces a watched call with another call to the specified callback function and then caches the modified fragment. The callback function must have the same signature as the watched call, so all parameter and return types must match. An initial list of calls to watch are invoked in the `init_syscall` function, but as will be explained shortly, the enforcement scheme was modified to allow watch calls from anywhere in the policy code, a necessity for the enforcement of adaptive policies. The callback function for the file open system call in the policy in figure 6 is the `myopen` function, as specified by the `watch` function called in `init_syscall`. The policy author is responsible for preserving the semantics of the target code, so re-routed system calls must be handled appropriately in the callback function. This especially applies to instances when a call does not produce a security violation, and the program must continue to run. The policy author must make the call to the original rerouted call from within the callback function if this is the case. Line 17 in figure 6 is an example of such a call. Note the use of the functions `void strata_syscallback_begin(unsigned)` and `void strata_syscallback_end (unsigned)` in same figure. They bracket the policy code that makes a call to the original watched system call `SYS_open`. In order to prevent this call from being rerouted back to the callback `myopen` function, the policy author uses these functions to tell Strata to ignore this particular call. Doing so prevents the policy code from entering an infinite loop and halting further execution. This is another responsibility of the policy author to write correct policies and not alter the semantics of the original program. The previous example used a system call watch to illustrate certain points. A policy involving a ge-

neric call would employ the same format; only the corresponding API functions `watch_call(void *func, void *callback), unwatch_call(void *func), void strata_callback_begin (void *),` and `void strata_callback_end (void *)` would be used instead.

Watch table

In order for the fragment builder to keep track of all the calls it must watch while mediating the execution of an un-trusted binary, it uses a data structure called the Watch Table. The Watch Table is simply a *hash table*, which is a specific kind of data structure used commonly in computer science mostly for is ability to quickly insert and retrieve data entries. This situation is ideal for our purposes, because we do not need to manipulate the data in any other way than simply storing it (e.g. sort) to enforce our policies. During translation, the fragment builder consults the table when ever a system or generic call is encountered, and the call is rewritten to a corresponding callback function if necessary. This modified fragment is then written to the fragment cache for subsequent execution on the processor.

Adaptive Policies

The focus of this project was to further extend dynamic policy flexibility by giving policy authors even more options for policy specification. *Adaptive policy enforcement* was the result of this extension. The driving force for developing these policies was the idea to allow a watch to be declared at any point in the policy code as opposed to solely in the `init_syscall` function. By allowing watches to be declared in preexisting callback functions, the policy has the ability to respond to certain calls by watching new ones. This in effect changes the policy rules. The new watches incur callback invocations with new policy code that would otherwise have not been possible to execute. Adaptive policies can also disable watches. The `unwatch` functions provided in the API modify the Watch Table to remove specified watch entries. Expenses incurred at both translation and execution times are motives for providing adaptive policy behavior. Certain watches only have to be in place when a security threat is more likely to occur. Providing adaptive behavior allows policy authors to first justify a security threat before enforcing an expensive policy. The following example, shown in Figure 7, again manipulates file-open system calls.

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <strata.h>
 4  #include <sys/syscall.h>
 5
 6  int myopen1 (const char *path, int oflag) {
 7          char absfilename[1024];
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <strata.h>
 4  #include <sys/syscall.h>
 5
 6  int new_myopen (const char *path, int oflag) {
 7          char absfilename[1024];
 8          int fd;
 9
10          strata_syscallback_begin(SYS_open);
```

```
11
12          makepath_absolute(absfilename,path,1024);
13          if (strncmp(absfilename,"/etc/",5) == 0) {
14             strata_fatal("Request Not Allowed!");
15          }
16          fd = syscall(SYS_open, path, oflag);
17
18          strata_syscallback_end(SYS_open);
19
20          return fd;
21  }
22
23  int myopen (const char *path, int oflag) {
24          char absfilename[1024];
25          int fd;
26
27          strata_syscallback_begin(SYS_open);
28
29          makepath_absolute(absfilename,path,1024);
30          if (strcmp(absfilename,"/etc/passwd") == 0) {
31             (*TI.unwatch_syscall)(SYS_open);
32             strata_flush();
33             (*TI.watch_syscall)(SYS_open, new_myopen);
34          }
35          fd = syscall(SYS_open, path, oflag);
36
37          strata_syscallback_end(SYS_open);
38
39          return fd;
40  }
41
42  void init_syscall() {
43          (*TI.watch_syscall)(SYS_open, myopen);
44  }
45
46  int main(int argc, char *argv[]) {
47          FILE *f;
48          int i = 1;
49          strata_init();
50          init_syscall();
51          strata_start();
52
53          while (i < argc) {
54            if (argc<2 || (f=fopen(argv[i],"r")) == NULL) {
55               fprintf(stderr,"Can't open file.\n");
56               exit(1);
57            }
58            ++i;
59          }
60          strata_stop();
61          return 0;
62  }
```

Figure 7. File Open Adaptive Policy.

Initially the `SYS_open` system call is translated to transfer control to the `myopen` callback function. Within `myopen`, the policy code tests to determine if the file in question is the `/etc/passwd` file. If it is, the callback function for `SYS_open` is changed to `new_myopen`.

A new policy is enforced then for each subsequent call of `SYS_open`. The new policy is defined in `new_myopen` to deny access to the entire `etc/` directory. In this particular case, the same system call was used in both the old and new policy. The new policy can watch a completely different set of system calls if desired. Changing policies is accomplished simply by declaring watches as a response to some event defined in an old policy.

Changing the policy does incur some expense. Since all watches and their callback functions are cached, any modification then requires the cached code to be updated. Currently the only way to do this is to flush the cache and re-translate the policy and target code. The previous example uses this method. Adaptive policies are then beneficial when the re-translation cost is at least equal to the savings gained by delaying enforcement of a computationally expensive policy until a security threat is justified. Figure 8 shows an alternative method to write the same policy just presented.

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <strata.h>
 4  #include <sys/syscall.h>
 5
 6  static int policy_step = 0;
 7
 8  int myopen (const char *path, int oflag) {
 9          char absfilename[1024];
10          int fd;
11
12          strata_syscallback_begin(SYS_open);
13
14          makepath_absolute(absfilename,path,1024);
15
16          switch (policy_step) {
17                  case 0:
18                  if (strcmp(absfilename,"/etc/passwd") == 0) {
19                     policy_step = 1;
20                     fd = syscall(SYS_open, "fake.c", oflag);
21                     printf("Fileopen blocked.\n");
22                  } else {
23                     fd = syscall(SYS_open, path, oflag);
24                  }
25                  break;
26
27                  case 1:
```

```
28                  if (strncmp(absfilename,"/etc/",5) == 0) {
29                     strata_fatal("Request Not Allowed!");
30                  } else {
31                     fd = syscall(SYS_open, path, oflag);
32                  }
33                  break;
34          }
35          strata_syscallback_end(SYS_open);
36
37          return fd;
38  }
39
40  void init_syscall() {
41          (*TI.watch_syscall)(SYS_open, myopen);
42  }
43
44  int main(int argc, char *argv[]) {
45          FILE *f;
46          int i = 1;
47          strata_init();
48          init_syscall();
49          strata_start();
50
51          while (i < argc) {
52             if (argc < 2 || (f = fopen(argv[i],"r")) == NULL) {
53                fprintf(stderr,"Can't open file.\n");
54                exit(1);
55             }
56             ++i;
57          }
58          strata_stop();
59          return 0;
60  }
```

Figure 8. Alternate File Open Adaptive Policy

Policies written in this manner never have to re-translate because all watches are defined in init_syscall. State variables are used to determine when a policy change is necessary. The variable `policy_step` in line 6 is an example of such a state variable. While the `SYS_open` system call only has one callback function, it still is subject to two different policies depending on the state of the system. We show you this policy, which again only uses one system call, for simplification purposes. When more watches are needed the format is exactly the same. Consider the policy in Figure 9 which prevents the target code from executing a shell after the user identification has been set to zero. This policy requires two system calls to be monitored, `SYS_execve` and `SYS_setuid`. Unlike previous examples, this policy also shows how our scheme is not forced to terminate execution

when a security violation occurs. Note that lines 24 through 25 simply ignore the `execve` request if the user identification, `curuid`, is zero.

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <unistd.h>
 4  #include <strata.h>
 5  #include <sys/syscall.h>
 6
 7  static int curuid = -1;
 8
 9  int mysetuid (int uid) {
10          int retval;
11
12          strata_syscallback_begin(SYS_setuid);
13          curuid = syscall(SYS_setuid, uid);
14          strata_syscallback_end(SYS_setuid);
15
16          return retval;
17  }
18
19  int myexecve (const char *path, char *const argv[],
20          char *const envp[]) {
21          int retval;
22          strata_syscallback_begin(SYS_execve);
23
24          if (curuid == 0) {
25             printf("Request Not Allowed!");
26          } else {
27             retval = syscall(SYS_execve, path, argv, envp);
28          }
29          strata_syscallback_end(SYS_execve);
30          return retval;
31  }
32
33  void init_syscall() {
34          (*TI.watch_syscall)(SYS_execve, myexecve);
35          (*TI.watch_syscall)(SYS_setuid, mysetuid);
36  }
37
38
39  int main (int argc, char *argv[]) {
40
41          char *args[2] = {"bin/sh",NULL};
42          char *envs[1] = {NULL};
43
44          strata_init();
45          init_syscall();
46          strata_start();
47
48          setuid(0);
49          execve("./bin/sh", args, envs);
50
51          strata_stop();
52
53          return 0;
54  }
```

Figure 9. Adaptive Policy to Prevent Root-Shell Execution

As mentioned previously, adaptive behavior is intended to provide alternative methods for enforcing policies that are inherently computationally expensive. This expense arises from either an extensive list of watches to translate, or expensive policy code executing in the callback functions. An example of the second, which adaptive behavior seems to have potential to improve performance is audit, or logging policies. The following example shown in Figure 10 is a policy that logs statistical information about file open system calls only after the `/etc/` directory has been accessed.

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <strata.h>
 4  #include <sys/syscall.h>
 5  #include <unistd.h>
 6  #include <sys/types.h>
 7  #include <sys/stat.h>
 8
 9  FILE *out;
10  static int policy_step = 0;
11
12  int myopen (const char *path, int oflag) {
13          char absfilename[1024];
14          int fd;
15          int st;
16          struct stat buf;
17
18          strata_syscallback_begin(SYS_open);
19
20          makepath_absolute(absfilename,path,1024);
21
22  switch (policy_step) {
23          case 0:
24          if (strcmp(absfilename,"/etc/passwd") == 0) {
25             policy_step = 1;
26             printf("Fileopen blocked...Log beginning...\n");
27                  fd = syscall(SYS_open, "fake.c", oflag);
28          } else {
29             fd = syscall(SYS_open, path, oflag);
30          }
31          break;
32          default:
33          st = stat(path, &buf);
34
35         fprintf(out, "File:
              %s.\nDevice ID:
              %d.\nSerial Number:
              %d.\nAccess Mode:
              %d.\nNumber of Links:
              %d.\nUser ID of file owner:
              %d.\nGroup ID of group owner:
              %d.\nFile size(bytes):
```

```
                %d.\nLast access time:
                %d.\nLast modification time:
                %d.\nLast file stat us change time:
                %d.\n\n\n", path, buf.st_dev, buf.st_ino,
                buf.st_mode, buf.st_nlink, buf.st_uid, buf.st_gid,
                buf.st_size, buf.st_atime, buf.st_mtime,
                uf.st_ctime);
36
37          fd = syscall(SYS_open, path, oflag);
38          break;
39  }
40          strata_syscallback_end(SYS_open);
41
42          return fd;
43  }
44
45  void init_syscall() {
46          (*TI.watch_syscall)(SYS_open, myopen);
47  }
48
49  int main(int argc, char *argv[]) {
50          FILE *f;
51          int i = 1;
52          strata_init();
53          out = fopen("log.out", "w");
54          init_syscall();
55          strata_start();
56
57          while (i < argc) {
58             if (argc < 2 || (f = fopen(argv[i],"r")) == NULL) {
59                fprintf(stderr,"Can't open file.\n");
60                exit(1);
61             }
62             ++i;
63          }
64          close(out);
65          strata_stop();
66          return 0;
67  }
```

Figure 10. Adaptive Audit Policy

Modifications to Strata

In order to facilitate the enforcement of our adaptive policies, the Watch Table had to be made accessible to the API at runtime. Specifically, the API had to be able to modify the table so new policies could insert or remove watches as policy code executed. As an extension to current hash table we added the `unwatch_syscall` and `unwatch_call` functions to the API. This required mostly target dependent code modifica-

tion (specifically the target interface TI and the target specific builder). The target independent code that was added involved only defining the functions.

A special *cache flush* function was needed to tell the fragment builder to flush the fragment cache via the security API. This function is part of the API and is named `strata_flush`. Again, most of the code involved was target dependent. The fragment builder was set to recognize the call to `strata_flush` in the policy code. Then a short SPARC assembly program had to be written that would carry out the flush, which is emitted into the fragment cache upon every occurrence. These changes to the Strata security extension application and the security API made possible the implementation of our adaptive policies. Refer to the Appendix for a listing of the targ-build.c file.

Summary of General Rules for Writing Policies

The current state of the security API forces the policy author to perform several steps when using certain functionality that will be automated in the future. The first is the initiation process for Strata. As explained in Chapter 4, function calls to `initiate`, `start`, and `stop` will not be necessary in the future as research continues on Strata. Also note that currently the policy code and the target code must be in the same executable binary for testing purposes. The target code may either be written in the `main` function or called from it. The policy code will be kept independent in the future as an important measure in ensuring execution transparency and the practicality of this application.

Recall that Strata enforces policies by merging the policy code with the target binary and caching it before it is actually executed. All subsequent execution then takes place from cached code. Adaptive policies generally respond to events by altering them-

selves to enforce new policy rules. In order for the new policy to be enforced, the policy code must again be merged with the target binary and cached. This fact requires that the policy author to manually perform a call to `strata_flush` in the security API that empties the fragment cache causing Strata to automatically retranslate the target binary to include the new policy.

The policy author is also expected to use the bracket functions properly. Since the fragment builder ignores this code, security could be compromised if this policy code itself were written with malicious intent. We assume that policy authors do not have malicious intent; however functionality is in place to only permit the use of the bracket functions from within policy code. This prevents a malicious user from using these functions in the target code in attempt to evade the policy being enforced.

Callback functions must have the same signature as their associated call. This is so that all parameters passed to the watched call can be passed to the callback function for use in the policy code.

Evaluation of the Security API

The API supported in the C programming language provides a great deal of flexibility for the policy author. Writing policies is exactly like writing a C program, with the advantage of having additional functionality to watch system and generic call instances. This of course does propose some challenge to the author to learn the C language. The potential for implementing any needed policy seems to depend on the author's knowledge of C rather than the SDT application's ability to provide a policy's feasibility which can be viewed as an advantage due to the great amount of documentation on the C language

and its intimate relationship with the UNIX operating system. Such a learning curve is much more gradual than one presented by numerous other policy enforcement schemes, which introduce an entirely new language for policy specification. One disadvantage to our scheme is that currently, policy authors are required to have a detailed knowledge of security concepts to implement a robust policy. For instance, a policy to prevent file writes requires the author to know which system calls are used in this process, and a robust policy would have to account for all the ways a system could write a file, which could be dozens depending on the architecture of the target system. This places a great deal of responsibility on the author to think like a hacker and consider even remote possibilities for policy subversion. Otherwise the system could be compromised and any efforts for policy enforcement would be futile. Other schemes try to avoid this challenge by creating their own specification languages that include predefined methods to robustly implement certain basic security functions. In this case the language would include a method to prevent a file write that can be called by the author without him knowing the lower-level operations involved. Our current system does currently have this setback but the flexibility of the C language has the potential to overcome. Predefined security methods like the one described previously could be packaged in separate header files to be called by the policy, and because C is such a widely used language, these header files would be accessible to a wide audience of knowledgeable security professionals for creation, testing and maintenance. These security professionals could make use of the low-level functionality to produce methods for less security-aware policy authors. Additional functionality added to the SDT application could also help facilitate this process. In conclusion, the security API provided in our policy enforcement scheme gives a security

knowledgeable policy author the flexibility of low-level manipulation combined with potential high-level definitions to implement a wide range of dynamic and adaptive security polices with minimal programming effort.

A Note on the Security of Strata

It was the focus of this project to extend the flexibility of our user-specified policy enforcement scheme. It was not the focus however to make Strata itself, or our security extension application totally secure. Their currently exists a number of vulnerabilities that may allow policy code to be circumvented as well as actually disabling the operation of Strata. Providing low-level code safety is a crucial task for providing security. The process is reasonably straightforward however its implementation is beyond the scope of this project.

Chapter Six: Conclusion

Summary

The first goal of this thesis was to show that our current policy enforcement scheme could be extended to include the enforcement of adaptive policies giving policy authors even more policy definition flexibility. This goal was accomplished. Modifications were made to the security API and to the policy enforcement application built into the Strata SDT framework as described in Chapter Five.

The next goal was to evaluate this addition to the application through experimentation with a number of dynamic and adaptive polices. It is our conclusion that the system we have presented is beneficial to the security community. SDT technology allows enormous flexibility in policy specification, as does the nature of our API which is supported in the widely used C programming language. Policies can be applied to programs without the need of source code manipulation prior to execution, which aids in the degree of execution transparency. This capability alone transcends the options presented by static policy enforcement. Our system already performs a derivative of execution monitoring without the need of operating system assistance, and unlike many static solutions, we are not limited to the enforcement of a single policy. The dynamic information provided at run-time allows dynamic and adaptive policies to be written that at least equal, if not supercede, the scope of any static policy and require significantly less programmer effort. Our enforcement mechanism, Strata, makes all of this possible.

The enforcement of adaptive policies only increases policy flexibility by giving policy authors the ability to increase or decrease the degree of security a policy applies

during execution. By responding to pre-defined events, an adaptive policy can justify threats before enforcing computationally expensive policies. While this savings in performance has not been fully quantified due to the wide range of policies our system may potentially support, by simply providing new options to security engineers the probability of attaining security solutions increases.

Interpretations

The problem of computer security is far from being solved. This system does not claim that it totally solves it because no scheme can make a system perfectly secure due to the extensively large variety of security threats that exist in the real world. This fact then makes it very difficult to evaluate our scheme without real world exposure. Despite this fact, we are confident that our system provides a robust policy enforcement scheme that is easily extendable to meet future demands of the security community.

User-specified policy enforcement in general means that the user is given explicit control over the security measures taken on the system. Our policy enforcement scheme allows this control to be taken to another level, through the mediation of the binary stream of a running program. Adaptive policy functionality then attempts to further increase the flexibility in policy specification. There is certainly a question of how useful this application is currently to the security ignorant user. An adaptive policy can become complicated fairly quickly especially when the author considers all possible paths of actions that could be taken and situations that could occur during execution. Further complications arise in policy specification because of low-level system call manipulation. Common users do not generally have detailed knowledge in this area, which would limit

their ability to implement effective policies. In order for our system to be useful to this user, the execution transparency of the enforcement mechanism must be supported by pre-defined, certified, and easily attainable security policies.

Recommendations

Our system is not currently ready for release because low-level code safety is not guaranteed. Code safety is essential for preventing a user from subverting the operation of Strata and compromising the system.

As mentioned earlier, the performance benefits of using adaptive policies have not been fully quantified. This is of special interest for those which can be written in alternative ways (recall the policy which incurs less cache flushes). Our performance evaluation was limited to the SPEC benchmark which is not sensible for testing the types of policies we developed. Other benchmarks which incorporate more file access will be necessary to accurately assess the performance of our policies.

Finally, more policies need to be written to test our scheme. Our original goal was to extend the security application to create even more specification flexibility in our scheme. Now that this has been done, it is up to knowledgeable security professionals to implement realistic policies and produce viable security solutions.

Bibliography

Software Dynamic Translation

1. V. Bala, S. Banerjia, and E. Duesterwald. *Dynamo: a transparent dynamic optimization system*. In SIGPLAN '00 Conference on Programmng Language Design and Implementation, pages 1-12, 2000.

2. Ebcioglu K., and Altman, E.R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In Proceedings of the 24th Annual International Symposium on Computer Architecture. 26-37.

3. Kevin Scott and Jack Davidson. Strata: A Software Dynamic Translation Infrastructure. Technical Report CS-2001-17, Department of Computer Science, University of Virginia, July 2001.

4. Kevin Scott, Jack Davidson, and Kevin Skadron. Low-Overhead Software Dynamic Translation. Technical Report CS-2001-18, Department of Computer Science, University of Virginia, July 2001.

Security Policy Enforcement

5. Tim Fraser, Lee Badger, and Mark Feldman. *Hardening COTS Software with Generic Software Wrappers*. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 1999.

6. David Evans and Andrew Twyman. *Flexible policy-directed code safety*. In IEEE Security and Privacy, Oakland, May 1999.

7. Ulfar Erlingsson and Fred B. Schneider. *SASI enforcement of security policies: A retrospective*. In Proceedings of the 1999 New Security Paradigms Workshop, Caledon Hills, September 1999.

8. P. Devanbu and S. Stubblebine. *Software engineering for security: A roadmap*. In A. Finkelstein, editor, The Future of Software Engineering. ACM Press, New York, 2000.

9. Laurence Cholvy and Frederic Cuppens, "*Analyzing Consistency of Security Policies*," in Proceedings of the 1997 IEEE Symposium on Security and Privacy, pp. 103-112, IEEE Computer Society Press, Los Alamitos, CA, 1997.

10. F. Cuppens and C. Saurel. *Specifying a Security Policy: A Case Study*. In Proc. of the computer security foundations workshop, Kenmare, Co. Kerry, Ireland, 1996.

11. Kevin Scott and Jack Davidson. Software Security using Software Dynamic Translation, Technical Report CS-2001-29, Department of Computer Science, University of Virginia, November 2001.

Intrusion Detection

12. Ghosh, A.K., A. Schwartzbard, M. Schatz, *Learning Program Behavior Profiles for Intrusion Detection*", Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring, April 9-12, 1999, Santa Clara, CA.

13. K Jain and R Sekar. *User-level infrastructure for system call interposition: A platform for intrusion detection and confinement*. In ISOC Network and Distributed System Security, 2000.

14. C. Warrender, S. Forrest, and B. Pearlmutter. *Detecting intrusions using system calls: alternative data models*. In Proceedings of the 1999 IEEE Symposium on Security and Privacy, pages 133--145. IEEE Computer Society, 1999.

Anti-Hacking

15. Smith, Nathan P. *Stack Smashing Vulnerabilities in the UNIX Operating System [online*]. Available WWW: <http://reality.sgi.com/nate/machines/security/nate-buffer.ps> (1997).

16. Tsai, Timothy and Navjot Singh. *Libsafe: Protecting Critical Elements of Stacks*. White Paper 3-21-01, Avaya Labs, Avaya Inc. February, 2001.

17. Hashii, Manoj Lal, Raju Pandey, and Steven Samorodin. *Securing Systems Against External Programs*. IEEE Internet Computing, 2(6):35-- 45, Nov-Dec 1998. 13

18. Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

19. David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In Proceedings of the 2001 USENIX Security Symposium, 2001.

Code Safety

20. Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. *Efficient Software -Based Fault Isolation*. In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 203--216, December 1993.

21. Dexter Kozen. *Efficient code certification*. Technical Report 98-1661, Cornell University, Department of Computer Science, 1998. Available from http://www.cs.cornell.edu/kozen/secure.

22. George Necula. *Proof-carrying code*. In Conference Record of the ACM Symposium on Principles of Programming Languages. ACM Press, January 1996.

Computer Architecture

23. System V Application Binary Interface, SPARC Processor Supplement, Third Edition.

Appendix

Targ-build.c

Modified target dependent strata code to incorporate adaptive policy enforcement.

```
/*
 * build.c - SPARC fragment builder (fast)
 *
 * Copyright (c) 2000, 2001 - J. Kevin Scott and Jack W. Davidson
 *
 * This file is part of the Strata dynamic code modification infrastructure.
 * This file may be copied, modified, and redistributed for non-commercial
 * use as long as all copyright, permission, and nonwarranty notices are
 * preserved, and that the distributor grants the recipient permission for
 * further redistribution as permitted by this notice.
 *
 * Please contact the authors for restrictions applying to commercial use.
 *
 * THIS SOURCE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
 * MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 * Author: J. Kevin Scott
 * e-mail: jks6b@cs.virginia.edu
 * URL   : http://www.cs.virginia.edu/~jks6b
 *
 * $Id: targ-build.c,v 1.5 2001/10/17 14:24:04 jks6b Exp $
 *
 * $Log: targ-build.c,v $
 * Revision 1.5  2001/10/17 14:24:04  jks6b
 * Changes for version 1.0 pre 2.  Added target interface support, a SPARC
 * disassembler, miscellaneous debug support, support for self-modifying code,
 * etc.
 *
 * Revision 1.4  2001/10/02 19:36:18  jks6b
 * Added target interface.
 *
 * Revision 1.3  2001/06/11 23:01:08  jks6b
 * Made number of branch target address lsbs to discard in IBTC lookups a
 * run-time settable parameter (STRATA_THROW_AWAY).  It defaults to 2 lsbs
 * which are always going to be 0 on the SPARC.
 *
 * Revision 1.2  2001/06/11 21:21:08  jks6b
 * Added code to fast return mode that bypasses the call return block
 * when the delay slot instruction overwrites the return address register.
 * We now handle the case where %o7 is overwritten by a restore or a
 * "mov %g1, %o7".  This fixes all of SPECint2k.  However, we should make
 * the code more general.
 *
 * Revision 1.1.1.1  2001/06/11 01:54:33  jks6b
 * Initial CVS import of the Strata tree for version 1.0 release testing.
 *
 *
 */

#include <stdio.h>
#include <assert.h>
#include <strata.h>
#include "insn.h"

/* Watch stuff. */
#define WATCH_SYSCALL  0
#define WATCH_CALL     1
#define WATCH_TAB_SIZE 31
```

```
typedef struct watch_st {
        int ty;                /* Type of watch: WATCH_SYSCALL, WATCH_CALL */
        union {
                void *func;    /* Function call to watch */
                unsigned num;  /* Syscall number to watch */
        } u;
        int ignore;            /* If flag is set, ignore this watch */
        void *callback;                /* The callback to invoke */
        struct watch_st *next;
} watch;
watch *watch_tab[WATCH_TAB_SIZE];

/* Temporary global location for holding branch target addresses. */
unsigned bta_loc;

/* IBTC and FAST RETURN option flags and parameters. */
#define IBTC_THRESHOLD_DEFAULT 4
static int targ_opt_ibtc;
static int targ_ibtc_mask;
static int targ_throw_away;
static int targ_opt_fast_return;
static int targ_fast_return_first;
static int targ_opt_trace_flow;
#define DEFAULT_BT_ALIGN 0
static int targ_bt_align;

/* External function declarations. */
extern void targ_reenter (unsigned PC);
extern void targ_reenter_flush (unsigned PC);
extern char *sparcdis (char *buf, iaddr_t PC, insn_t insn);

/* Private function declarations. */
static int writes_PC (insn_t insn);
static void print_PC (iaddr_t PC);
static int make_mask(int n);
static void init_ibtc (strata_fragment *frag);
static watch *call_watch_lookup (void *func);
static watch *syscall_watch_lookup (unsigned num);
static iaddr_t targ_trap (strata_fragment *frag, iaddr_t PC, insn_t insn);
static int find_trap_num (strata_fragment *frag);
static void print_insn (iaddr_t PC, insn_t insn);

/* Size of trampolines (in instructions) for conditional branches. */
#define TRAMPSIZE 8

/* Print out a PC for debugging. */
void print_PC (iaddr_t PC) {
        fflush(stderr);
        fprintf(stderr,"%08x =>\n",PC);
}

/* Fetch an instruction for this target. */
insn_t targ_fetch (iaddr_t PC) {
        return *(insn_t *)PC;
}

/* Align PC as if it were a branch target. */
iaddr_t targ_branch_align (iaddr_t PC) {
        return (PC + targ_bt_align) & ~targ_bt_align;
}

/* A shorthand for strata_emit_insn() for the SPARC. */
#define EMIT(insn) strata_emit_insn(frag,insn,4);

/* Print an instruction to stderr. */
static void print_insn (iaddr_t PC, insn_t insn) {
        char buf[128];

        sparcdis(buf,PC,insn);
        fprintf(stderr,"%08x:  %08x  %s\n",PC,insn,buf);
}
```

```
/* Return the log-base-2 of n. */
static int make_mask(int n) {
        int mask;

        mask = 0;
        while(n != 1) {
                n = n >> 1;
                mask = (mask << 1) | 0x1;
        }
        return mask;
}

/* Allocate and initialize an IBTC for fragment frag. */
static void init_ibtc (strata_fragment *frag) {
        int i, *p;

        frag->targ_data = strata_allocate(stat_ibtc_size,FCACHE);

        #ifdef STATS
        stat_ibtc_mem += stat_ibtc_size;
        #endif


        p = (int *)frag->targ_data;
        for(i=0;i<2*stat_ibtc_nentries;i++) {
                *p++ = 0;
        }
}

/* Initialize the target dependent builder. */
void targ_init (void) {
        char *env;
        unsigned n, i;

        if (env = getenv("STRATA_IBTC_NENTRIES")) {
                targ_opt_ibtc = 1;
                stat_ibtc_nentries = atoi(env);
                stat_ibtc_size = stat_ibtc_nentries * 8;
                if (env = getenv("STRATA_IBTC_THRESHOLD")) {
                        stat_ibtc_threshold = atoi(env);
                } else {
                        stat_ibtc_threshold = IBTC_THRESHOLD_DEFAULT;
                }
                targ_ibtc_mask = make_mask(stat_ibtc_nentries);
                if (env = getenv("STRATA_THROW_AWAY")) {
                        targ_throw_away = atoi(env);
                } else {
                        targ_throw_away = 2;
                }
        } else {
                targ_opt_ibtc = 0;
                targ_ibtc_mask = 0;
                stat_ibtc_nentries = 0;
                stat_ibtc_size = 0;
                stat_ibtc_threshold = 0;
        }

        if (getenv("STRATA_FAST_RETURN")) {
                targ_opt_fast_return = 1;
                targ_fast_return_first = 0;
        } else {
                targ_opt_fast_return = 0;
        }

        if (getenv("STRATA_TRACE_FLOW")) {
                targ_opt_trace_flow = 1;
        } else {
                targ_opt_trace_flow = 0;
        }
```

```
        if (env = getenv("STRATA_BT_ALIGN")) {
                n = atoi(env);
        } else {
                n = DEFAULT_BT_ALIGN;
        }

        /* Compute the mask for the branch target alignment. */
        targ_bt_align = 0;
        for(i=0;i<n;i++) {
                targ_bt_align = (targ_bt_align << 1) | 0x1;
        }

        /* Initialize the watch table. */
        for(i=0;i<WATCH_TAB_SIZE;i++) {
                watch_tab[i] = NULL;
        }
}

/* Classify an instruction. */
unsigned targ_classify (insn_t insn) {
        unsigned op, op2, op3, dst, src1;

        op = (insn >> 30) & 0x3;
        switch(op) {
                case 0:
                        op2 = (insn >> 22) & 0x7;
                        if (op2 == 0 || op2 == 4) {
                                return STRATA_NORMAL;
                        } else {
                                return STRATA_PC_RELATIVE_BRANCH;
                        }
                case 1:
                        return STRATA_CALL;
                case 2:
                        op3 = (insn >> 19) & 0x3f;
                        switch(op3) {
                        case 0x36: case 0x37:
                                /* Implementation dependant instructions. */
                                /* !!!!!! Treating as normal ops !!!!!!   */
                                return STRATA_NORMAL;
                                break;
                        case 0x38:
                                dst = (insn >> 25) & 0x1f;
                                src1 = (insn >> 14) & 0x1f;
                                if (dst == G0 && (src1 == O7 || src1 == I7)) {
                                        return STRATA_RETURN;
                                } else {
                                        return STRATA_INDIRECT_BRANCH;
                                }
                                break;
                        case 0x39:
                                strata_fatal("Encountered SPARC V9 RETURN");
                                break;
                        case 0x3a: case 0x3b:
                                return STRATA_SPECIAL;
                                break;
                        default:
                                return STRATA_NORMAL;
                                break;
                        }
                case 3:
                        return STRATA_NORMAL;
                default:
                        assert(0);
        }
}

/* Handle a normal instruction -- copy insn to the next slot in frag, and
 * return the address of the next instruction to process.
 */
iaddr_t targ_normal (strata_fragment *frag, iaddr_t PC, insn_t insn) {
```

```
        if (strata_tracing) {
                print_PC(PC);
        }

        EMIT(insn);
        return PC + 4;
}

/* Handle a special instruction: right now, this lets us trap SPARC flushes */
iaddr_t targ_special (strata_fragment *frag, iaddr_t PC, insn_t insn) {
        unsigned op3;

        if (strata_tracing) {
                print_PC(PC);
        }

        op3 = (insn >> 19) & 0x3f;
        if (op3 == 0x3a) {
                return targ_trap(frag,PC,insn);
        } else if (op3 == 0x3b) {
                /* End fragment by calling targ_reenter_flush with PC+4 */
                EMIT(insn);
                EMIT(SAVE(SP,-96,SP));
                EMIT(SETHI(HI(PC+4),O0));
                EMIT(ORI(O0,LO(PC+4),O0));
                EMIT(CLR(O1));
                EMIT(SETHI(HI((unsigned)targ_reenter_flush),L0));
                EMIT(ORI(L0,LO((unsigned)targ_reenter_flush),L0));
                EMIT(JMPLI(L0,0,G0));
                EMIT(NOP);
                return 0;
        } else {
                assert(0);
        }
}

/* Handle trap instructions. */
static iaddr_t targ_trap (strata_fragment *frag, iaddr_t PC, insn_t insn) {
        unsigned cond, imm, rs1, rs2;
        int trapnum;
        watch *w;

        /* Decode the condition code. */
        cond = (insn >> 25) & 0xf;

        /* If not an unconditional trap, just emit it. */
        if (cond != 0x8) {
                EMIT(insn);
                return PC + 4;
        }

        /* Figure out if rs1+rs2 or rs1+imm */
        rs1 = (insn >> 14) & 0x3f;
        if ((insn >> 13) & 0x1) {
                imm = insn & 0x7f;

                /* If can't determine this is a software trap, pass through. */
                if (rs1 != G0 || imm != 0x8) {
                        EMIT(insn);
                        return PC + 4;
                }

                /* We have a software trap, figure out which. */
                trapnum = find_trap_num(frag);

                /* If we can't find the trap number, pass through. */
                if (trapnum == -1) {

                        /* Debug: Print out this fragment. */
                        targ_disassemble_fragment(frag);
```

```
                        print_insn(PC,insn);

                        EMIT(insn);
                        return PC + 4;
                }

                /* See if we're watching this trap. */
                w = syscall_watch_lookup(trapnum);
                if (w == NULL || w->ignore) {
                        EMIT(insn);
                        return PC + 4;
                } else {
                        return w->callback;
                }

        } else {
                rs2 = insn & 0x3f;

                /* Pass through right now. */
                EMIT(insn);
                return PC + 4;
        }
}

/* Search backwards in frag for the most recent def of %g1. */
static int find_trap_num (strata_fragment *frag) {
        insn_t insn;
        iaddr_t PC;
        unsigned imm;

        PC = frag->last_fPC - 4;
        while(PC >= frag->fPC) {
                insn = targ_fetch(PC);

                /* See if we have a "mov imm, %g1" */
                if ((insn & 0xffffe000) == 0x82102000) {
                        imm = insn & 0x1fff;
                        return imm;
                }


                PC -= 4;
        }

        return -1;
}

/* Handle a PC relative jump.  These will typically require two trampolines:
 * one for the taken target, and one for the not-taken target.
 */
iaddr_t targ_pcrel_branch (strata_fragment *frag, iaddr_t PC, insn_t insn) {
        unsigned op2, cond, trampsize;
        insn_t dsi;
        iaddr_t takenaddr, t_targ, nt_targ;
        strata_fragment *tf;

        if (strata_tracing) {
                print_PC(PC);
        }

        /* Do a bit of decoding. */
        op2 = (insn >> 22) & 0x7;

        /* Figure out the branch taken address. */
        switch(op2) {
        case 1:         /* branch on integer cc with prediction */
                takenaddr = PC + (SEXT19(insn & 0x7ffff) << 2);
                break;
        case 2:         /* branch on integer cc */
                takenaddr = PC + (SEXT22(insn & 0x3fffff) << 2);
                break;
```

```
case 3:        /* branch on contents of integer register */
       takenaddr = PC +
              (SEXT16((insn&0x3fff)|((insn>>6)&0xc000)) << 2);
       break;
case 5:        /* branch on float cc with prediction */
       takenaddr = PC + (SEXT19(insn & 0x7ffff) << 2);
       break;
case 6:        /* branch on float cc */
       takenaddr = PC + (SEXT22(insn & 0x3fffff) << 2);
       break;
default:
       fprintf(stderr,"Unrecognized branch %08x @ %08x\n",
                     insn,PC);
       strata_fatal("Unrecognized branch type");
}

/* Grab condition specifier. */
cond = (insn >> 25) & 0xf;

if (op2 != 3 && (cond == 8)) {

       /* Branch always.  We don't emit this instruction
        * into the fragment cache.  If its annul bit is
        * 1, we don't emit the delay slot instruction
        * either.  The next PC is at takenaddr.
        */

       /* Go on to next instruction if the annul bit set. */
       if ((insn >> 29) & 0x1) {
              return takenaddr;
       }

       /* Otherwise, emit the delay slot instruction.  */
       dsi = (*TI.fetch)(PC + 4);
       if (writes_PC(dsi))
              strata_fatal("Invalid delay slot instruction");
       EMIT(dsi);

       /* Go to next instruction */
       return takenaddr;

} else if (op2 != 3 && (cond == 0)) {

       /* Branch never.  This instruction acts like a nop.
        * We don't emit it.  And if its annul bit is set,
        * we don't even emit the delay slot instruction.
        * The next PC is PC + 8.
        */

       /* Go on to next instruction if the annul bit set. */
       if ((insn >> 29) & 0x1) {
              return PC + 8;
       }

       /* Otherwise, emit the delay slot instruction.  */
       dsi = (*TI.fetch)(PC + 4);
       if (writes_PC(dsi))
              strata_fatal("Invalid delay slot instruction");
       EMIT(dsi);

       /* Go to next instruction */
       return PC + 8;

} else {

       /* Conditional branch. */

       /* Figure out if the branch targets are already in
        * the fragment cache or not.  If they are, we
        * generate code that looks a bit different.
        */
```

```
tf = strata_lookup_fragment(takenaddr);
t_targ = tf ? tf->fPC : 0;

tf = strata_lookup_fragment(PC+8);
nt_targ = tf ? tf->fPC : 0;

/* Figure out how big the trampoline is based on
 * whether or not the not-taken target is present in
 * the fragment cache.
 */

if (nt_targ) {
       trampsize = 2;
} else {
       trampsize = TRAMPSIZE;
}

/* Rewrite the branch. */

if (op2 == 1 || op2 == 5) {
       EMIT(REWRITE_BRANCH19(insn,trampsize+2));
} else if (op2 == 2 || op2 == 6) {
       EMIT(REWRITE_BRANCH22(insn,trampsize+2));
} else {
       EMIT(REWRITE_BRANCH16(insn,trampsize+2));
}

/* Emit the delay slot instruction.  */
dsi = (*TI.fetch)(PC+4);
if (writes_PC(dsi))
       strata_fatal("Invalid delay slot instruction");
EMIT(dsi);

if (!nt_targ) {

       /* If the not taken successor block is not
        * in the fragment cache, emit a full blown
        * trampoline that gives control back to
        * Strata at its terminus.
        */

       /* Record fixup for the not taken trampoline. */
       strata_patch_later(frag->last_fPC,PC+8,PATCH_TRAMP);

       /* Emit the not taken trampoline. */
       EMIT(SAVE(SP,-96,SP));
       EMIT(SETHI(HI(PC+8),O0));
       EMIT(ORI(O0,LO(PC+8),O0));
       EMIT(CLR(O1));
       EMIT(SETHI(HI((unsigned)targ_reenter),L0));
       EMIT(ORI(L0,LO((unsigned)targ_reenter),L0));
       EMIT(JMPLI(L0,0,G0));
       EMIT(NOP);

} else {

       if (strata_tracing) {
       fflush(stderr);
       fprintf(stderr,"Branch directly to %08x.\n",nt_targ);
       }

       /* Not taken target is in F$.  Jump to it. */
       EMIT(BA((nt_targ - frag->last_fPC)>>2));
       EMIT(NOP);

}

if (!t_targ) {

       /* If the taken successor block is not
```

```
                         * in the fragment cache, emit a full blown
                         * trampoline that gives control back to
                         * Strata at its terminus.
                         */

                        /* Record a fixup for the taken trampoline. */
                        strata_patch_later(frag->last_fPC,takenaddr,PATCH_TRAMP);

                        /* Emit the taken trampoline. */
                        EMIT(SAVE(SP,-96,SP));
                        EMIT(SETHI(HI(takenaddr),O0));
                        EMIT(ORI(O0,LO(takenaddr),O0));
                        EMIT(CLR(O1));
                        EMIT(SETHI(HI((unsigned)targ_reenter),L0));
                        EMIT(ORI(L0,LO((unsigned)targ_reenter),L0));
                        EMIT(JMPLI(L0,0,G0));
                        EMIT(NOP);

                } else {

                        if (strata_tracing) {
                        fflush(stderr);
                        fprintf(stderr,"Branch directly to %08x.\n", t_targ);
                        }

                        /* Not taken target is in F$.  Jump to it. */
                        EMIT(BA((t_targ - frag->last_fPC)>>2));
                        EMIT(NOP);
                }

                /* Set type on this fragment. */
                frag->ty = STRATA_FRAG_CBRANCH;

                return 0;
        }

}

/* Handle a call. */
iaddr_t targ_call (strata_fragment *frag, iaddr_t PC, insn_t insn) {
        strata_fragment *ret_frag;
        watch *w;
        insn_t dsi;
        iaddr_t addr, targ_PC;
        int dsi_writes_o7;

        if (strata_tracing) {
                print_PC(PC);
        }

        /* We want to partially inline calls.  This means eliminating
         * the call instruction itself and emitting instructions that
         * emulate the side effects of the call
         */

        /* Figure out where this call goes in the original program. */
        targ_PC = PC + (insn << 2);

        /* Figure out if we need to inline a callback attached to this call. */
        if ((w = call_watch_lookup(targ_PC)) != NULL) {
                if (!w->ignore)
                        targ_PC = w->callback;
        }

        /* Fetch the delay slot instruction. */
        dsi = (*TI.fetch)(PC+4);

        /* Determine whether or not the delay slot is a restore. */
        dsi_writes_o7 = 0;
        if (dsi == 0x9e100001) {
                dsi_writes_o7 = 1;
```

```
} else if (((dsi>>30)&0x3) == 0x2) {
        if (((dsi>>19)&0x3f) == 0x3d) {
                /* Restore. */
                dsi_writes_o7 = 1;
        }
}

/* Side effect 1: Place PC of call into %o7. */
if (targ_opt_fast_return && targ_PC != (unsigned)strata_stop) {

        /* Determine if return fragment is in the FCACHE. */

        if (strata_tracing) {
        fprintf(stderr,"Looking up return fragment %08x\n",PC+8);
        }

        ret_frag = strata_lookup_fragment(PC+8);
        if (!ret_frag && !dsi_writes_o7) {

                /* If the return fragment is not in the FCACHE, we
                 * need to force the builder to put it there and
                 * patch PC of that fragment into the assignment to
                 * %o7 we emit.
                 */

                if (strata_tracing) {
                fprintf(stderr,"Return fragment not found\n");
                }

                strata_enqueue_address(PC+8);
                strata_patch_later(frag->last_fPC,PC+8,PATCH_RETADDR);
                EMIT(SETHI(HI(0),O7));
                EMIT(ORI(O7,LO(0),O7));

        } else if (ret_frag) {

                /* The return fragment is in the FCACHE.  Place its
                 * address into %o7.
                 */

                if (strata_tracing) {
                fprintf(stderr,"Return fragment found\n");
                }

                EMIT(SETHI(HI(ret_frag->fPC-8),O7));
                EMIT(ORI(O7,LO(ret_frag->fPC-8),O7));
        }

} else {
        /* For normal returns, put a text return addr into O7. */
        EMIT(SETHI(HI(PC),O7));
        EMIT(ORI(O7,LO(PC),O7));
}

/* Side effect 2: The delay slot instruction.  */

/* Bail out if the delay slot instruction is a branch or call. */
if (writes_PC(dsi))
        strata_fatal("Invalid delay slot instruction");

/* Don't write the delay slot instruction if it is a NOP. */
if (dsi != NOP) {
        EMIT(dsi);
}


/* If this is a call to strata_flush, then we need to emit
 * the sequence of code into the fragment cache to save
 * the state and flush the cache.
 */
if (targ_PC == (unsigned)strata_flush) {
```

```
                EMIT(SAVE(SP,-96,SP));
                EMIT(SETHI(HI(PC+8),O0));
                EMIT(ORI(O0,LO(PC+8),O0));
                EMIT(CLR(O1));
                EMIT(SETHI(HI((unsigned)targ_reenter_flush),L0));
                EMIT(ORI(L0,LO((unsigned)targ_reenter_flush),L0));
                EMIT(JMPLI(L0,0,G0));
                EMIT(NOP);
        }

        /* If this is a call to strata_stop, then we need to emit
         * the sequence of code into the fragment cache to release
         * Strata's hold on the program.  Then, we jump to the
         * top of the block just formed.  When execution reaches
         * the end, control will be returned to the application
         * until strata_start is called again.
         */
        if (targ_PC == (unsigned)strata_stop) {
                /* Emit code to set strata_good_stop to 1. */
                EMIT(SETHI(HI((unsigned)&strata_good_stop),O0));
                EMIT(ORI(O0,LO((unsigned)&strata_good_stop),O0));
                EMIT(LDUWI(O0,0,O1));
                EMIT(ADDI(O1,1,O1));
                EMIT(STWI(O1,O0,0));

                /* Emit code to transfer to strata_stop. */
                EMIT(SETHI(HI((unsigned)strata_stop),O0));
                EMIT(ORI(O0,LO((unsigned)strata_stop),O0));
                EMIT(JMPLI(O0,0,G0));
                EMIT(NOP);
                return 0;
        }

        if (strata_tracing) {
                fprintf(stderr,"CALL target is %08x\n",targ_PC);
        }

        return targ_PC;
}

/* Handle a return: these can be handled as indirect branches on the SPARC. */
iaddr_t targ_return (strata_fragment *frag, iaddr_t PC, insn_t insn) {
        iaddr_t ra;

        if (strata_tracing) {
                fprintf(stderr,"RETURN Instruction Found\n");
                print_PC(PC);
        }

        /* Figure out the return mode we're using and do the right thing. */
        if (targ_opt_fast_return) {
                if (targ_fast_return_first) {
                        targ_fast_return_first = 0;
                        ra = targ_ind_branch(frag,PC,insn);
                } else {
                        /* For fast returns, return to FCACHE caller. */
                        EMIT(insn);
                        EMIT((*TI.fetch)(PC + 4));
                        ra = 0;
                }
        } else {
                /* For normal returns, handle as indirect branch. */
                ra = targ_ind_branch(frag,PC,insn);
        }

        /* Set the type on this fragment, and return. */
        frag->ty = STRATA_FRAG_RET;
        return ra;
}

/* Handle an indirect branch. */
```

```
iaddr_t targ_ind_branch (strata_fragment *frag, iaddr_t PC, insn_t insn) {
        strata_fragment *ret_frag;
        unsigned dst;
        insn_t dsi;

        if (strata_tracing) {
                print_PC(PC);

        }

        /* Set the fragment type to indirect branch. */
        frag->ty = STRATA_FRAG_IBRANCH;

        /* Emit code to compute branch target address into bta_loc. */

        /* Save application contents of %o0 & %o1. */
        EMIT(STWI(O0,SP,-4));  /* st %o0, [%sp-4] */
        EMIT(STWI(O1,SP,-8));  /* st %o1, [%sp-8] */

        /* Compute the branch target into %o0. */
        EMIT(CHANGE_TO_ADD(insn,O0));

        /* Compute the address of the bta temp location. */
        EMIT(SETHI(HI((unsigned)(&bta_loc)),O1));
        EMIT(ORI(O1,LO((unsigned)(&bta_loc)),O1));

        /* Store the computed branch target in temporary loc. */
        EMIT(STWI(O0,O1,0));   /* st %o0, [%o1] */

        /* Restore the application state. */
        EMIT(LDUWI(SP,-4,O0)); /* ld [%sp-4], %o0  */
        EMIT(LDUWI(SP,-8,O1)); /* ld [%sp-8], %o1  */

        /* Emit code to emulate the side effects of jmpl or return. */

        /* Write the jmpl PC to the destination register unless the
         * destination is %g0.  If the destination is %o7 and we're
         * in fast return mode, then we need to write the fragment
         * cache PC (-8) to which the return fragment has been
         * mapped.
         */
        dst = (insn >> 25) & 0x1f;
        if (targ_opt_fast_return && dst == O7) {
                ret_frag = strata_lookup_fragment(PC+8);
                if (!ret_frag) {
                        strata_enqueue_address(PC+8);
                        strata_patch_later(frag->last_fPC,PC+8,PATCH_RETADDR);
                        EMIT(SETHI(HI(0),O7));
                        EMIT(ORI(O7,LO(0),O7));
                } else {
                        EMIT(SETHI(HI(ret_frag->fPC-8),O7));
                        EMIT(ORI(O7,LO(ret_frag->fPC-8),O7));
                }
        } else if (dst != G0) {
                /* For normal returns, put a text return addr into O7. */
                EMIT(SETHI(HI(PC),dst));
                EMIT(ORI(dst,LO(PC),dst));
        }

        /* Emit the delay slot instruction. */
        dsi = (*TI.fetch)(PC + 4);
        if (writes_PC(dsi))
                strata_fatal("Invalid delay slot instruction");

        /* Don't write the delay slot instruction if it is a NOP. */
        if (dsi != NOP) {
                EMIT(dsi);
        }

        /* Now, emit the indirect branch trampoline. */
```

```
        /* Allocate ourselves a buffer frame. */
        EMIT(SAVE(SP,-96,SP));

        /* Compute the address of the bta temp location. */
        EMIT(SETHI(HI((unsigned)(&bta_loc)),O1));
        EMIT(ORI(O1,LO((unsigned)(&bta_loc)),O1));

        /* Get the branch target address into %o0. */
        EMIT(LDUWI(O1,0,O0));  /* ld [%o1], %o0 */

        if (targ_opt_ibtc) {

                if (frag->targ_data == NULL)
                        init_ibtc(frag);

                /* Emit code to lookup the branch target in the IBTC */

                /* Load address of the IBTC into %o2. */
                EMIT(SETHI(HI((unsigned)(frag->targ_data)),O2));
                EMIT(ORI(O2,LO((unsigned)(frag->targ_data)),O2));

                /* Use bta (in %o0) to hash into IBTC. */
                EMIT(SRLI(O0,targ_throw_away,O1));
                EMIT(ANDI(O1,targ_ibtc_mask,O1));
                EMIT(SLLI(O1,3,O1));   /* Index into IBTC in %o1. */
                EMIT(ADD(O1,O2,O2));   /* Address of IBTC element in %o2. */
                EMIT(LDUWI(O2,0,O1));  /* Tag in %o1. */
                EMIT(LDUWI(O2,4,O3));  /* Translation in %o3. */
                EMIT(RDCCR(O2));       /* Store condition codes in %o2. */
                EMIT(CMP(O0,O1));      /* Compare bta and tag. */
                EMIT(BNE(4));          /* Branch if not equal to tramp. */
                EMIT(WRCCR(O2));       /* Restore condition codes. */
                EMIT(JMPLI(O3,0,G0));  /* Jump to translated address. */
                EMIT(RESTORE);         /* Restore on our way out. */

        }

        /* Emit code to bounce into builder. */

        /* Load fragment address into %o1. */
        EMIT(SETHI(HI((unsigned)frag),O1));

        EMIT(ORI(O1,LO((unsigned)frag),O1));

        /* Bounce off trampoline to finish context switch. */
        EMIT(SETHI(HI((unsigned)targ_reenter),L0));
        EMIT(ORI(L0,LO((unsigned)targ_reenter),L0));
        EMIT(JMPLI(L0,0,G0));
        EMIT(NOP);

        return 0;
}

/* Patch location patch_loc so that it refers to targPC */
void targ_patch (unsigned patch_loc, unsigned targPC, int ty) {

        if (strata_tracing) {
                fflush(stderr);
                fprintf(stderr,"Fixup location=%08x with value=%08x, ",
                        patch_loc, targPC);
        }

        switch(ty) {
        case PATCH_TRAMP:
                if (strata_tracing) {
                        fprintf(stderr,"type = PATCH_TRAMP\n");
                }
                (*TI.emit)(BA((targPC-patch_loc)>>2),patch_loc);
                (*TI.emit)(NOP,patch_loc+4);
                (*TI.flush)(patch_loc,patch_loc+8);
```

```
                break;

        case PATCH_RETADDR:
                if (strata_tracing) {
                        fprintf(stderr,"type = PATCH_RETADDR\n");
                }
                (*TI.emit)(SETHI(HI(targPC-8),O7),patch_loc);
                (*TI.emit)(ORI(O7,LO(targPC-8),O7),patch_loc+4);
                (*TI.flush)(patch_loc,patch_loc+8);
                break;
        default:
                strata_fatal("Unrecognized patch type");
        }

}

/* Write instruction directly to location addr. */
void targ_emit (insn_t insn, iaddr_t addr) {
        char buf[128];

        if (strata_tracing) {
                print_insn(addr,insn);
        }

        *(iaddr_t *)addr = insn;
}

/* Handle a mispredicted indirect branch. */
void targ_indirect_branch_miss (strata_fragment *from, strata_fragment *to) {
        unsigned i, *p;

        if (targ_opt_ibtc) {

                /* Hash to->PC into the IBTC. */
                i = ((to->PC >> targ_throw_away) & targ_ibtc_mask) << 1;
                p = (unsigned *)from->targ_data + i;

                /* Update the entry. */
                *p       = to->PC;
                *(p + 1) = to->fPC;

        }
}

/* Return 1 if the instruction writes the PC, i.e., is a call, return, jmpl, */
/* or branch of some sort.  Otherwise return a 0.                             */
static int writes_PC (insn_t insn) {
        unsigned auxop;

        switch( (insn >> 30) & 0x3) {
        case 0:
                auxop = (insn >> 22) & 0x7;
                if (auxop == 0 || auxop == 4) {
                        /* illtrap, sethi, or nop */
                        return 0;
                } else {
                        /* branch */
                        return 1;
                }
        case 1:
                /* call */
                return 1;
        case 2:
                auxop =  (insn >> 19) & 0x3f;
                if (auxop == 0x38 || auxop == 0x39) {
                        /* jmpl or return */
                        return 1;
                } else {
                        return 0;
                }
        case 3:
```

```
                /* load or store */
                return 0;
        }
        strata_fatal("Doh! This can't happen!");
}

/* Called after fragment is allocated, but before any code is inserted. */
void targ_begin_fragment(strata_fragment *frag) {
}

/* Called before fragment contents are flushed and finalized. */
void targ_end_fragment(strata_fragment *frag) {
}

/* Disassemble fragment frag and print to stdout. */
void targ_disassemble_fragment(strata_fragment *frag) {
        iaddr_t PC;
        insn_t insn;

        fprintf(stderr,"Fragment %08x => %008x\n",frag->PC,frag->fPC);
        PC = frag->fPC;
        while(PC < frag->last_fPC) {
                insn = targ_fetch(PC);
                print_insn(PC,insn);
                PC += 4;
        }
}

/* Install a system call watch.  When syscall num executes, invoke callback. */
void targ_watch_syscall(unsigned num, void *callback) {
        watch *w;
        unsigned h;

        /* Allocate new watch. */
        NEW(w,PERM);
        w->ty = WATCH_SYSCALL;
        w->u.num = num;
        w->ignore = 0;
        w->callback = callback;

        /* Compute hash index. */
        h = num % WATCH_TAB_SIZE;

        /* Link into the appropriate hash chain. */
        w->next = watch_tab[h];
        watch_tab[h] = w;
}

/* Uninstall a system call watch. */
void targ_unwatch_syscall(unsigned num) {
        watch *w, *prev;
        unsigned h;

        /* Compute hash index. */
        h = num % WATCH_TAB_SIZE;

        /* Link into the appropriate hash chain. */
        for(prev=NULL,w=watch_tab[h];w!=NULL;prev=w,w=w->next) {
                if (w->ty == WATCH_SYSCALL && w->u.num == num) {
                        if (prev == NULL) {
                                watch_tab[h] = watch_tab[h]->next;
                        } else {
                                prev->next = w->next;
                        }
                }
        }

        /* Flush the fragment cache. */
}

/* Uninstall a call watch. */
```

```
void targ_unwatch_call(void *func) {
	unsigned h;
	watch *w, *prev;

	/* Compute hash index. */
	h = (unsigned)func % WATCH_TAB_SIZE;

	/* Link into the appropriate hash chain. */
	for(prev=NULL,w=watch_tab[h];w!=NULL;prev=w,w=w->next) {
		if (w->ty == WATCH_CALL && w->u.func == func) {
			if (prev == NULL) {
				watch_tab[h] = watch_tab[h]->next;
			} else {
				prev->next = w->next;
			}
		}
	}

	/* Flush the fragment cache. */
}

/* Install a call watch.  When call func executes, invoke callback. */
void targ_watch_call(void *func, void *callback) {
	watch *w;
	unsigned h;

	/* Allocate new watch. */
	NEW(w,PERM);
	w->ty = WATCH_CALL;
	w->u.func = func;
	w->ignore = 0;
	w->callback = callback;

	/* Compute hash index. */
	h = (unsigned)func % WATCH_TAB_SIZE;

	/* Link into the appropriate hash chain. */
	w->next = watch_tab[h];
	watch_tab[h] = w;
}

static watch *call_watch_lookup (void *func) {
	watch *w;
	unsigned h;

	/* Compute hash index. */
	h = (unsigned)func % WATCH_TAB_SIZE;

	for(w=watch_tab[h];w!=NULL;w=w->next) {
		if (w->ty == WATCH_CALL && w->u.func == func) {
			return w;
		}
	}
	return NULL;
}

static watch *syscall_watch_lookup (unsigned num) {
	watch *w;
	unsigned h;

	/* Compute hash index. */
	h = num % WATCH_TAB_SIZE;

	for(w=watch_tab[h];w!=NULL;w=w->next) {
		if (w->ty == WATCH_SYSCALL && w->u.num == num) {
			return w;
		}
	}

	return NULL;
}
```

```
/* This is probably not the place for this stuff -- refactor later. */

/* Disable watches on func until strata_callback_end. */
void strata_callback_begin (void *func) {
        watch *w;

        w = call_watch_lookup(func);
        if (w == NULL) {
                fprintf(stderr,"Function %08x is not being watched.\n",func);
                strata_fatal("Error in strata_callback_begin()");
        }
        w->ignore = 1;
}

/* Reenable watches on func. */
void strata_callback_end (void *func) {
        watch *w;

        w = call_watch_lookup(func);
        if (w == NULL) {
                fprintf(stderr,"Function %08x is not being watched.\n",func);
                strata_fatal("Error in strata_callback_end()");
        }
        w->ignore = 0;
}

/* Disable watches on syscall num until strata_syscallback_end. */
void strata_syscallback_begin (unsigned num) {
        watch *w;

        w = syscall_watch_lookup(num);
        if (w == NULL) {
                fprintf(stderr,"Syscall %d is not being watched.\n",num);
                strata_fatal("Error in strata_syscallback_begin()");
        }
        w->ignore = 1;
}

/* Reenable watches on syscall num. */
void strata_syscallback_end (unsigned num) {
        watch *w;

        w = syscall_watch_lookup(num);
        if (w == NULL) {
                fprintf(stderr,"Syscall %d is not being watched.\n",num);
                strata_fatal("Error in strata_syscallback_end()");
        }
        w->ignore = 0;
}
```

Dynamic and Adaptive Policy Examples

Fileopen.c

```
#include <stdio.h>
#include <string.h>
#include <strata.h>
#include <sys/syscall.h>


void makepath_absolute(char *absfilename,char *path, int length) {

        char *ptr;
        int i = 0;
        int l = 0;

        ptr = &(*path);
        while (*ptr != NULL && l < length) {
                absfilename[i] = *ptr;
                *ptr++;
                i++;
        }
}

int myopen (const char *path, int oflag) {
        char absfilename[1024];
        int fd;
        char *ptr;
        int i = 0;

        strata_syscallback_begin(SYS_open);

        makepath_absolute(absfilename,path,1024);
        printf("\n%s\n", absfilename);

        if (strcmp(absfilename,"/etc/passwd") == 0) {
                strata_fatal("Strata Policy does not allow this request!");
        }
        fd = syscall(SYS_open, path, oflag);

        strata_syscallback_end(SYS_open);

        return fd;
}

void init_syscall() {
        (*TI.watch_syscall)(SYS_open, myopen);
}

int main(int argc, char *argv[]) {
        FILE *f;
        strata_init();
        init_syscall();
        strata_start();
        if (argc < 2 || (f = fopen(argv[1],"r")) == NULL) {
                fprintf(stderr,"Can't open file.\n");
                exit(1);
        }
        printf("File %s opened.\n",argv[1]);

        strata_stop();
        return 0;
}
```

Diropen.c

```
#include <stdio.h>
#include <string.h>
```

```
#include <strata.h>
#include <sys/syscall.h>


void makepath_absolute(char *absfilename,char *path, int length) {

        char *ptr;
        int i = 0;
        int l = 0;

        ptr = &(*path);
        while (*ptr != NULL && l < length) {
                absfilename[i] = *ptr;
                *ptr++;
                i++;
        }
}

int myopen (const char *path, int oflag) {
        char absfilename[1024];
        int fd;

        strata_syscallback_begin(SYS_open);

        makepath_absolute(absfilename,path,1024);
        printf("\n%s\n", absfilename);

        if (strncmp(absfilename,"/etc/",5) == 0) {
                strata_fatal("Strata Policy does not allow this request!");
        }
        fd = syscall(SYS_open, path, oflag);

        strata_syscallback_end(SYS_open);

        return fd;
}

void init_syscall() {
        (*TI.watch_syscall)(SYS_open, myopen);
}

int main(int argc, char *argv[]) {
        FILE *f;
        strata_init();
        init_syscall();
        strata_start();
        if (argc < 2 || (f = fopen(argv[1],"r")) == NULL) {
                fprintf(stderr,"Can't open file.\n");
                exit(1);
        }
        printf("File %s opened.\n",argv[1]);

        strata_stop();
        return 0;
}
```

Fdopen.c

```
#include <stdio.h>
#include <string.h>
#include <strata.h>
#include <sys/syscall.h>


void makepath_absolute(char *absfilename,char *path, int length) {

        char *ptr;
        int i = 0;
        int l = 0;

        ptr = &(*path);
```

```
        while (*ptr != NULL && l < length) {
                absfilename[i] = *ptr;
                *ptr++;
                i++;
        }
        absfilename[i] = '\0';
}

int myopen1 (const char *path, int oflag) {

        char absfilename[1024];
        int fd;

        strata_syscallback_begin(SYS_open);

        makepath_absolute(absfilename,path,1024);
        printf("\n%s\n", absfilename);

        if (strncmp(absfilename,"/etc/",5) == 0) {
                strata_fatal("Strata Policy does not allow this request!");
        }
        fd = syscall(SYS_open, path, oflag);

        strata_syscallback_end(SYS_open);

        return fd;
}

int myopen (const char *path, int oflag) {
        char absfilename[1024];
        int fd;

        strata_syscallback_begin(SYS_open);

        makepath_absolute(absfilename,path,1024);
        printf("\n%s\n", absfilename);

        if (strcmp(absfilename,"/etc/passwd") == 0) {
                (*TI.unwatch_syscall)(SYS_open);
                strata_flush();
                (*TI.watch_syscall)(SYS_open, myopen1);
        }
        fd = syscall(SYS_open, path, oflag);

        strata_syscallback_end(SYS_open);

        return fd;
}

void init_syscall() {
        (*TI.watch_syscall)(SYS_open, myopen);
}

int main(int argc, char *argv[]) {
        FILE *f;
        int i = 1;
        strata_init();
        init_syscall();
        strata_start();

        while (i < argc) {
                if (argc < 2 || (f = fopen(argv[i],"r")) == NULL) {
                        fprintf(stderr,"Can't open file.\n");
                        exit(1);
                }
                printf("File %s opened.\n",argv[i]);
                ++i;
        }
        strata_stop();
        return 0;
}
```

Fdopen2.c

```
#include <stdio.h>
#include <string.h>
#include <strata.h>
#include <sys/syscall.h>

static int policy_step = 0;

void makepath_absolute(char *absfilename,char *path, int length) {
        char *ptr;
        int i = 0;

        ptr = &(*path);
        while (*ptr != NULL && i < length) {
                absfilename[i] = *ptr;
                *ptr++;
                i++;
        }
        absfilename[i] = '\0';
}

int myopen (const char *path, int oflag) {
        char absfilename[1024];
        int fd;
        strata_syscallback_begin(SYS_open);

        makepath_absolute(absfilename,path,1024);
        printf("path = %s\n", absfilename);

        switch (policy_step) {
                case 0:
                if (strcmp(absfilename,"/etc/passwd") == 0) {
                        policy_step = 1;
                        fd = syscall(SYS_open, "fake.c", oflag);
                        printf("Fileopen blocked by policy.\n");
                } else {
                        fd = syscall(SYS_open, path, oflag);
                }
                break;

                case 1:
                if (strncmp(absfilename,"/etc/",5) == 0) {
                        strata_fatal("Strata Policy does not allow this request!");
                } else {
                        fd = syscall(SYS_open, path, oflag);
                }
                break;
        }


        strata_syscallback_end(SYS_open);

        return fd;
}

void init_syscall() {
        (*TI.watch_syscall)(SYS_open, myopen);
}

int main(int argc, char *argv[]) {
        FILE *f;
        int i = 1;
        strata_init();
        init_syscall();
        strata_start();

        while (i < argc) {
                if (argc < 2 || (f = fopen(argv[i],"r")) == NULL) {
                        fprintf(stderr,"Can't open file.\n");
```

```
                        exit(1);
                }
                printf("File %s opened.\n",argv[i]);
                ++i;
        }
        strata_stop();
        return 0;
}
```

Fdopen4.c

```
#include <stdio.h>
#include <string.h>
#include <strata.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

FILE *out;
static int policy_step = 0;

void makepath_absolute(char *absfilename,char *path, int length) {
        char *ptr;
        int i = 0;

        ptr = &(*path);
        while (*ptr != NULL && i < length) {
                absfilename[i] = *ptr;
                *ptr++;
                i++;
        }
        absfilename[i] = '\0';
}

int myopen (const char *path, int oflag) {
        char absfilename[1024];
        int fd;
        int st;
        struct stat buf;

        strata_syscallback_begin(SYS_open);

        makepath_absolute(absfilename,path,1024);
        printf("path = %s\n", absfilename);

switch (policy_step) {
        case 0:
        if (strcmp(absfilename,"/etc/passwd") == 0) {
                policy_step = 1;
                printf("Fileopen blocked by policy...Log beginning...\n");
/*              fd = syscall(SYS_open, "fake.c", oflag);*/
        } else {
                fd = syscall(SYS_open, path, oflag);
        }
        break;

        default:
        st = stat(path, &buf);

        fprintf(out, "File: %s.\nDevice ID: %d.\nSerial Number: %d.\nAccess Mode:
%d.\nNumber of Links: %d.\nUser ID of file owner: %d.\nGroup ID of group owner: %d.\nFile
size(bytes): %d.\nLast access time: %d.\nLast modification time: %d.\nLast file status
change time: %d.\n\n\n", path, buf.st_dev, buf.st_ino, buf.st_mode, buf.st_nlink,
buf.st_uid, buf.st_gid, buf.st_size, buf.st_atime, buf.st_mtime, buf.st_ctime);

        fd = syscall(SYS_open, path, oflag);
        break;
}

        strata_syscallback_end(SYS_open);
```

```
        return fd;
}

void init_syscall() {

        (*TI.watch_syscall)(SYS_open, myopen);
}

int main(int argc, char *argv[]) {
        FILE *f;
        int i = 1;
        strata_init();
        out = fopen("log.out", "w");
        init_syscall();
        strata_start();

        while (i < argc) {
                if (argc < 2 || (f = fopen(argv[i],"r")) == NULL) {
                        fprintf(stderr,"Can't open file.\n");
                        exit(1);
                }
                printf("File %s opened.\n",argv[i]);
                ++i;
        }
        close(out);
        strata_stop();
        return 0;
}
```

Preventexec.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <strata.h>
#include <sys/syscall.h>

static int curuid = -1;

int mysetuid (int uid) {
        int retval;

        strata_syscallback_begin(SYS_setuid);
        curuid = syscall(SYS_setuid, uid);

        strata_syscallback_end(SYS_setuid);

        return retval;
}

int myexecve (const char *path, char *const argv[],
        char *const envp[]) {
        int retval;
        strata_syscallback_begin(SYS_execve);

        if (curuid == 0) {
                strata_fatal("Naughty, naughty");
        } else {
                retval = syscall(SYS_execve, path, argv, envp);
        }
        strata_syscallback_end(SYS_execve);
        return retval;
}

void init_syscall() {
        (*TI.watch_syscall)(SYS_execve, myexecve);
        (*TI.watch_syscall)(SYS_setuid, mysetuid);
}
```

```
int main (int argc, char *argv[]) {
        FILE *f;
        char *args[2] = {"hole",NULL};
        char *envs[1] = {NULL};

        strata_init();
        init_syscall();
        strata_start();

        setuid(0);
        execve("./hole", args, envs);

        strata_stop();

        return 0;
}
```

Combo.c

```
#include <stdio.h>
#include <string.h>
#include <strata.h>
#include <sys/syscall.h>

static int fileopened = 0;
static int curuid = -1;

int mysetuid (int uid);

int myopen (const char *path, int oflag) {

        char absfilename[1024];
        int fd;
        printf("open watched...\n");

        (*TI.watch_syscall)(SYS_setuid, mysetuid);
        strata_flush();
        strata_syscallback_begin(SYS_open);
        fd = syscall(SYS_open, path, oflag);
        strata_syscallback_end(SYS_open);

        return fd;
}

int myexecve (const char *path, char *const argv[],
 char *const envp[]) {
        int retval;
        printf("execve watched...");

        strata_syscallback_begin(SYS_execve);
        retval = syscall(SYS_execve, path, argv, envp);
        strata_syscallback_end(SYS_execve);
        return retval;
}

void init_syscall() {

        (*TI.watch_syscall)(SYS_open, myopen);
        (*TI.watch_syscall)(SYS_execve, myexecve);
}

int main(int argc, char *argv[]) {

        FILE *f;
        char *args[2] = {"bin/sh", 0};

        strata_init();
        init_syscall();
        strata_start();

        setuid(0);
```

```
        if (argc < 2 || (f = fopen(argv[1],"r")) == NULL) {
                fprintf(stderr,"Can't open file.\n");
                exit(1);
        }
        printf("File %s opened.\n",argv[1]);

        setuid(0);
        execve("bin/sh", args);

        strata_stop();
        return 0;
}


int mysetuid (int uid) {
        int retval;
        printf("setuid watched...\n");

        strata_syscallback_begin(SYS_setuid);
        curuid = syscall(SYS_setuid, uid);
        strata_syscallback_end(SYS_setuid);

        return retval;
}
```

Cookies.c

```
#include <stdio.h>
#include <string.h>
#include <strata.h>
#include <sys/syscall.h>
#include "snarf.h"

static int socket_fd = -1;

/* Copy src buffer to dst removing cookies*/
int remove_cookies(char *dst, const void *src,
 int size);

/* Callback for the so_socket system call. */
int my_so_socket (int a, int b, int c, char *d,
 int e) {
        int fd;

        strata_syscall_begin(SYS_so_socket);
        /* Make the system call and record the */
        /* file descriptor */
        socket_fd = syscall(SYS_so_socket,a,b,c,d,e);
        strata_syscall_end(SYS_so_socket);

        return socket_fd;
}

/* Callback for the write system call */
int my_write (int fd, void *buf, int size) {
        char new_buf[1024];
        int s, new_size;

        strata_syscall_begin(SYS_write);
        /* Only look at writes to socket_fd
        /* and only rewrite HTTP headers. */
        if (fd == socket_fd &&
         (new_size = remove_cookies(new_buf,buf,size)))
                s = syscall(SYS_write,fd,new_buf,new_size);
        else
                s = syscall(SYS_write,fd,buf,size);
        strata_syscall_end(SYS_write);

        return s;
}
```

```
void init_syscall() {
       (*TI.watch_syscall)(SYS_so_socket,my_so_socket);
       (*TI.watch_syscall)(SYS_write,my_write);
}

int main(int argc, char *argv[]) {
       snarf_main(argc, argv);
}
```

Limitpack.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <string.h>
#include <strata.h>
#include <sys/syscall.h>

#define RATE 10000
#define TOPRATE 10000000
#define DISCARD_PORT 9999
#define PAYLOAD_SIZE 1024

void xmit (const char *host, int nbytes);

static int socket_fd = -1;

/* Compute the delay necessary to maintain */
/* the desired rate */
int limiting_delay (double rate, time_t tbeg,time_t tend, int last_len, int len);

/* Callback for the so_socket call */
int my_so_socket (int a,int b,int c,char *d,int e) {
       int fd;

       strata_syscallback_begin(SYS_so_socket);
       /* Make the system call and */
       /* record the file descriptor */
       socket_fd = syscall(SYS_so_socket,a,b,c,d,e);
       strata_syscallback_end(SYS_so_socket);

       return fd;
}

/* Callback for the write system call */
int my_send (int s, const void *msg, size_t len,
 int flags) {
       int result;
       time_t now;
       static int last_len = 0;
       static time_t last_time = 0;

       strata_syscallback_begin(SYS_send);
       /* Only look at writes to socket_fd */
       /* and only rewrite HTTP headers */
       if (s == socket_fd) {
              now = time(NULL);
              sleep(limiting_delay(RATE,last_time, now,len,last_len));
              last_len = len;
              last_time = now;
       }
       result = syscall(SYS_send,s,msg,len,flags);
       strata_syscallback_end(SYS_send);

       return result;
}
```

```
void init_syscall() {
       (*TI.watch_syscall)(SYS_so_socket,my_so_socket);
       (*TI.watch_syscall)(SYS_send,my_send);
}

main(int argc, char *argv[]) {
       if (argc == 3)
              xmit(argv[1],atoi(argv[2]));
       else
              fprintf(stderr,"Usage: %s host nbytes\n",argv[0]);
}

/* Transmit nbytes to discard port (9) on host */
void xmit (const char *host, int nbytes) {
       int sd, bytes_sent;
       struct sockaddr_in sin;
       struct sockaddr_in pin;
       struct hostent *hp;
       char *payload[PAYLOAD_SIZE];
       time_t begin, elapsed;
       double rate;

       /* go find out about the desired host machine */
       if ((hp = gethostbyname(host)) == 0) {
              perror("gethostbyname");
              exit(1);
       }

       /* fill in the socket structure with host info */
       memset(&pin, 0, sizeof(pin));
       pin.sin_family = AF_INET;
       pin.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;
       pin.sin_port = htons(DISCARD_PORT);

       /* grab an Internet domain socket */
       if ((sd = socket(AF_INET,SOCK_STREAM, 0)) == -1) {
              perror("socket");
              exit(1);
       }

       /* connect to PORT on HOST */
       if (connect(sd, (struct sockaddr *) &pin,
              sizeof(pin)) == -1) {
              perror("connect");
              exit(1);
       }

       begin = time(0);
       bytes_sent = 0;
       while(bytes_sent < nbytes) {
              /* send a message to the server PORT */
              /* on machine HOST */
              if (send(sd,payload,sizeof(payload),0) == -1) {
                     perror("send");
                     exit(1);
              }
              bytes_sent += sizeof(payload);
              printf(".");
              fflush(stdout);
       }

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

Thesis Proposal

UNDERGRADUATE THESIS PROJECT PROPOSAL

School of Engineering and Applied Science
University of Virginia

ADAPTIVE SECURITY POLICIES
ENFORCED BY SOFTWARE DYNAMIC TRANSLATION

Submitted By
Paul H. Lamanna
Computer Science

TCC 401
Section 3 (11 AM)
October 21, 2001

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in TCC Courses.

_____

Approved _____ Date _____
Technical Advisor – Kevin Skadron

Approved _____ Date _____
TCC Advisor – Ingrid Townsend

Table of Contents

1. Abstract

This document is a proposal to construct a Software Dynamic Translation (SDT) application with the ability to enforce an adaptable security policy. Security policy enforcement is inherently computationally expensive, so ideally policies should only be enforced when a security threat is fully justified or they should be custom tailored to each system in order to minimize performance overhead. SDT technology has the ability to monitor and/or modify the binary execution stream of a process while it is running, and thus supports an enormous potential for success with this application. This research will attempt to show that an SDT system has the ability to enforce an adaptable security policy by alternating the enforcement of policies in an arbitrary set of policies. Once this concept is proved, an iterative process will begin in an attempt to make these policies useful to the computer security community. Using this technology to implement an adaptable policy enforcement scheme is an economically and practically feasible project, and it is hoped that the flexibility and portability of such a tool will aid in a successful widespread distribution in the future.

2. Rationale and Objectives

Software Dynamic Translation (SDT) technology is a relatively recent innovation. In short, SDT has the ability to monitor and modify the native binary execution stream of a program while it is running, and it can do so transparently without the need of pre-compilation or programmer assistance. Implemented entirely in software, SDT systems run between a process and its host CPU, interrupting the execution stream immediately before it reaches the processor. By doing so, they achieve a capability to control execution dynamically. It is important to emphasize that a run-time perspective of an executing process has unique and exciting benefits. Various implementations of SDT have surfaced over the past few years, realizing concepts such as binary translation and dynamic optimization. Binary translation technology allows for the portability of programs between differing architectures. By translating a binary stream from one instruction set architecture to another, a program can run on hardware other than what was originally intended. Dynamic optimization technology actually improves the

performance of programs while they are running, in contrast to a compiler which performs its optimizations statically at compile time. Perhaps what is so impressive about these technologies is that they accomplish all of this without ever having to manipulate the program at any time prior to execution. This advantage is not only valuable to the portability and flexibility of these applications but the catalyst of this thesis project. Due to this unique capability, SDT technology has an enormous potential for success in the field of computer security.

Computer security is becoming an increasingly important issue in this age of global communication and commerce and the Internet. It is highly researched for several reasons, its relevance to consumer privacy and due to the persistence and resourcefulness of attackers. Hackers are constantly discovering vulnerabilities in systems leaving behind a legacy of malicious programs known as exploits. As long as hackers have been hard at work causing problems, there have been efforts to prevent the damage they cause. These efforts have been both preventative of attacks and responsive to them, statically and dynamically. One such application, developed at the University of Virginia by Professor Dave Evans and Dave Larochelle, statically analyzes C programs for a certain type of exploit [18]. This would be an example of a preventative measure, but also a very static solution, since the source code is required before compilation and the questionable code is never run. A similar task is also accomplished dynamically (a dynamically linked library) at run-time by a program developed at Avalya Labs named Libsafe [17, 15]. This application is also preventative, and accomplishing the same end goal, but doing so in a different manner. Another portion of the these efforts have evolved into the field known as intrusion detection systems (IDS), which as the name suggests, strives to

profile the behavior of normal systems so that attacks may be detected while taking place. This approach is drastically different from the earlier examples and is a good indication of the diversity of computer security applications. Consider yet another method, the idea of enforcing user defined security polices. This approach offers a great deal of flexibility because policies can be tailored to provide different levels of security for different platforms. Policies are most commonly defined as a set of safety properties that place constraints resource operations [6]. By controlling computing resources and being suspicious of how processes use them, a policy may prevent attacks by reducing the mobility of malicious code.

Different users have a need for differing security applications, with different levels of security sensitivity. Certainly, the Department of Defense building computer network would require a greater number of security measures than a personal webpage. On a general level though, users value their privacy and security but are extremely sensitive to a decline in the performance of their machines. All security applications produce some performance overhead, but in order to achieve widespread use they must operate with some degree of transparency, to provide the maximum protection possible with the least amount of user interaction required and performance penalties incurred. The dynamic capabilities of SDT could potentially aid this degree of transparency and offer new approaches for current security practices.

I plan to implement an SDT application for enforcing security policies like those described previously, only the policies will have the ability to adapt to changing security requirements during the execution of one or more processes. Such an application will

have the ability to turn on or off its security monitoring or increase or decrease its level sensitivity based on the current state of the system. The application will:

- *support the concept of execution transparency,*
- *support the flexibility and portability of the policies by enforcing them in a platform independent manner,*
- *minimize user input concerning the definition of policies by making the policy adaptive to changing security requirements, and*
- *minimize performance overhead associated with enforcing the security policies.*

A major motivation for having adaptable security policies is that enforcing security policies is inherently computationally expensive. An expensive policy should only be enforced when it is considered necessary, and SDT technology will make this transition possible. Recent research in policy enforcement describes applications based in the operating system kernel, since system calls are readily accessible from this point [5]. SDT allows such system call access in addition to the rest of the execution stream, so policy enforcement with SDT will not need the aid of the operating system. Separating this functionality from the OS is ideal, because policies involving more than system call monitoring would be difficult or impossible to enforce by the kernel. Simply put, designing a kernel policy enforcement infrastructure fixes the type of policies that are enforceable. As a result, policy enforcement could complicate the kernel implementation which could introduce bugs, make maintenance more complicated, and even decrease performance.

3. Preliminary Impact Statement

Introduction

Any advance made in any area of computer security produces numerous impacts. These impacts affect not only the common user but also global networks of shared information, corporations, and even the hackers who invoke the threat. When security methods are improved, users gain an improved sense of security. This new environment nurtures progress. Whether the field is communication, commerce, or research, increased security allows professionals to concentrate on their work rather than worrying about its exploitation. It is important, however, that users do not gain a false sense of security, by overestimating the capabilities of security products. Such a scenario produces negative effects caused by users who place themselves in vulnerable situations they otherwise would not consider. Therefore, it is imperative that the scope of security software is well documented and tested, so that users are fully aware of its protective abilities and limitations.

Successful security systems fulfill four general requirements. They are 1) well documented, 2) lightweight (satisfactory performance on a given system), 3) easy to set up and maintain, and 4) and successful at defending a system from all of the attacks advertised. Notice that these requirements are associated not only with the level of protection provided by the product but with how the product affects the performance of the system. Recall that SDT systems have an enormous potential fulfill these requirements, resulting in a high probability of success.

Impacts to the Common User

The average computer user has little idea of the risks involved in everyday use. And even the more informed user would not be able to detect that an attack is taking place that simply violates privacy and causes no noticeable harm. An application such as the one proposed offers the common user a simple and transparent mechanism to deal with security issues so they do not have to. Since polices can be expressed more generally, less user input is required, making the product that much more attractive to the uninformed user, who is unfortunately the most commonly attacked. Also since the proposed policy enforcement scheme is implemented outside of the operating system it aids to the flexibility and portability of the tool.

Most applications available, especially those that enforce security policies, require a great deal of information about the system and to function properly. This information might not be readily available to the common user who, regardless of this fact, is still forced to make important decisions about the security of the system. One specific example of this is PC firewall technology. A firewall is simply a choke point in a network where traffic may be monitored and manipulated, and a PC firewall is designed to implement a firewall for individual hosts (the common user). Even though they are designed with non-expert users in mind, the firewall must still be configured regularly, in response to new adaptations of security threats. This requires knowledge of what the user wants to be protected against. In contrast, the tool proposed would aim to further decouple this requirement from security applications by allowing a common user to specify policies dealing with specific resources on the system rather than specific attack signatures. Functionality such as this suits a common user well because system specific requirements are more static, and more comprehendible than the alternative.

Impacts to the Security Engineer

The security engineer works to increase the level, and sense, of security on a larger scale than just a single machine. Worldwide communication networks and large corporate networks have a greater need for security measures, and also have more funding to allocate to more heavyweight security products. This fact lessens the impact of this application for security engineers, but the fact remains security advancements promote more security research. This research must be tested through distribution and constant use, and it is security engineers who often perform this task.

The major attraction of this tool for the security engineer is its flexibility for implementing dynamic and hopefully adaptable security policies. This introduces a wide range of possibilities for security engineers to combat the never-ending challenge of computer security. A security engineer will, in contrast to the common user, have a detailed knowledge of the system and the current state of security. With the dynamic content made available by this tool, he will be able to construct a wide variety of security policies specifically tailored to the system, the attacks in question, and to reduce performance overhead. While the common user also has this ability, this larger system offers more opportunity for diverse application due to a much greater number of users and a higher probability of security breaches. It is in this forum of research that hopefully the tool will be thoroughly tested and information and policies themselves may be passed down to the common user.

Impacts to the Hacker

The obvious impact is that this application might make it more difficult to break in to the system of a common user and the companies employing the security engineers. A deterrent such as this might cause the rate of successful attacks to decline, but could also provoke the hacker to write new exploits, searching for ways to get around the new barrier. As history has shown, they will more than likely find holes to exploit, and begin the iterative exchange between the security engineer and the hacker of successful attacks for new software patches. This will inevitably lead to an even more secure tool in time, but will incur the costs of compromised software.

Different hackers have different intentions for the damage they cause. For some, the damage is intended to make a point for some cause, and others it is simply intended for amusement. In either case, the resourcefulness of hackers must not be underestimated. It is important to understand that new security applications affect the hacker just as much as they provide a sense of security to those who use them. Therefore, new security applications are usually the most susceptible to attack, and must be made as invisible to the hacker as possible to lessen the negative effects of its deployment.

4. Review of Relevant Literature

This thesis project draws literature from several differing areas of study. A complete understanding of SDT is required to implement a new application for it. This knowledge not only includes SDT system design, but a great deal of underlying computer architectural principles. It is important to comprehend how a computer works from a low enough level of abstraction to be equal with the perspective of an SDT system (the assembly language level), and depending on which architecture is present this operation will differ from machine to machine. An understanding of machine level operation ties in intimately with computer security. Since the majority of security threats evolve from exploiting the low-level intricacies of a system, architectural knowledge is essential in undertaking security research. This practice of exploitation is known as *hacking*, and it is usually accomplished by any means necessary. The field of Computer Security attempts to discover, prevent, and respond to hacking. Like any other field, its goals are accomplished in various manners and through various abstractions, from low-level code certification to user and program behavior profiling to policy enforcement. All of these methods have proved to be effective to some degree in practice, but the resourcefulness of attackers presents a seemingly endless challenge for security engineers. The literature presented in the following sections will provide sufficient background, in SDT, computer architecture, and computer security, to enable the reader to fully comprehend the motivations behind this research and the call to arms for future research. With this foundation in place, an account of the additional concepts related directly to this project can be clearly presented.

An implementation of SDT, named *Strata*, is currently under construction at the

University of Virginia by the Computer Science PhD student James Kevin Scott. The Strata infrastructure provides a framework for implementing new SDT applications, so Strata has the ability to support functionality such as binary translation and dynamic optimization [4]. Hewlett Packard's Dynamo [1] is a transparent dynamic optimization tool which provides a similar framework, only code is also integrated to optimize PA-RISC binaries. Several other examples of dynamic optimization technology are Vulcan (IA-32), Mojo (IA-32), and DBT (PA-RISC). IBM's Daisy is an example of binary translation technology, translating the VLIW instruction set to PowerPC (Daisy also performs some dynamic optimization) [2]. FX!32 (IA-32 to Alpha), UQDBT (IA-32 to SPARC), and Transmeta's Code technology (IA-32 to VLIW), are other examples of binary translation technology. These products display the commercial success of SDT technology as well its flexibility to implement a wide range of services across multiple platforms with a single tool.

Both of the technical reports on Strata, entitled *Strata: A Software Dynamic Translation Infrastructure* and *Low-Overhead Software Dynamic Translation* give an excellent depiction of how Strata arbitrates program execution. Strata gains control of a process and maintains this control through special context switches, called trampolines, between its framework the running process. Each executed instruction is dynamically translated to become part of a cached sequence of code known as a fragment [4]. When the code is finally executed on the host processor it is done so from the fragment cache, a very fast memory resource containing the newly translated instruction. Note that this functionality provides a facility to modify this code before it is placed into the fragment

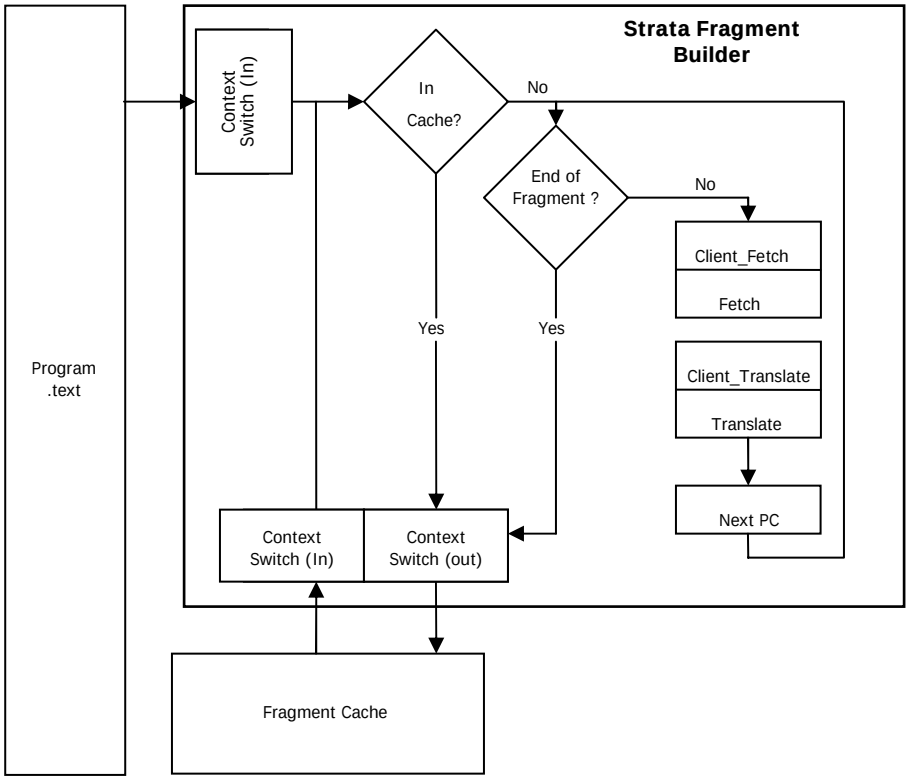


Figure1. Architecture of Strata.

cache, and before actual execution on the processor. The modification may be, and certainly not limited to, a code optimization or an insertion of security checking code. Figure 1. illustrates the architecture of Strata, specifically the formation and caching of fragments.

In order to define this thesis, I had to determine what modifications Strata should make to a program to make its operation more secure in some way. This task required sufficient background in the concepts of hacking and computer security. Generally speaking, hacking tends to exploit the low level intricacies of an operating system and/or architecture such as register organization, memory access and function calling and returning. A common, and maybe obsolete, example of hacking is a buffer overflow exploit or a stacking smashing attempt. These exploits take control of a process and force

it to execute malicious code [14]. Denial of service (DoS) attacks or distributed denial of service (DDoS) attacks contrastingly render a machine useless by bombarding its resources and overloading the CPU. The list goes on to include brute force attacks, IP spoofing, format string attacks, session hijacking, Trojan horses and many more. What is done to try and prevent these attacks, or to try and discover that they have happened and to take appropriate action, is the basis of computer security.

Various areas of computer security have evolved with these goals in mind. Each area seems to approach the task at hand through its own level of abstraction. Perhaps the lowest level of abstraction is pursued by a concept known as *code safety*. Code safety strives to provide control flow safety, memory safety, and stack safety to ensure programs behave properly [20]. Low-level code safety supports higher level security mechanisms, especially those that allow code modified with security checks to run. Without this type of safety in place a hacker could get around the security checks inserted to ensure that the code is secure. Software Fault Isolation [19] and Proof-Carrying Code [21] are two examples of mechanisms to provide low-level code safety. For any security extension to Strata to be successful in reality, low-level code safety must be ensured so that Strata's actual operation cannot be circumvented via hacking techniques. This task, however, is beyond the scope of this project, and will be left for future research.

One approach to dealing with the problems caused by hackers is to face the problem head on, constructing software that prevents specific types of attacks. LCLint statically analyzes C programs for buffer overflow exploits [18]. Libsafe is a dynamically linked library that performs bounds checking on certain C functions to prevent stack-smashing at run-time [17,15]. Strata, actually, has the ability to completely

prevent stack-smashing attempts with only several lines of code. This functionality along with a very simple policy enforcement mechanism is described in the technical report entitled *Low-Overhead Software Dynamic Translation*. Other efforts attempt to take a more general approach to the problem in the hopes of finding a more flexible solution to the continuously evolving hacking arsenal. Intrusion Detection Systems (IDS) is one such effort which, as the name suggests attempts to determine accurately when a system is under attack so that it may take proper action. It does this by profiling normal system behavior through system calls, user actions, and other past behavior. Any departure then from this normal behavior could be considered an attack [11]. A drawback of this approach though is that behavior profiles can never be one hundred percent accurate, meaning that often, innocent processes get treated as malicious ones. Yet another area tries to combat attacks by managing system resources through the use of security policies. Naccio is one such architecture which allows for the expression of security policies in a platform independent manner. These policies are then enforced by an application transformer, which rewrites the entire executable including wrappers around code that needs to be monitored by the policy [6]. There is research also in how to efficiently construct and place these wrappers, which are simple state machines that sit idle and listen for an event to occur. Upon this event the state machine takes action as defined by the policy [5]. The transformation invoked by Naccio is conceptually similar to how Strata translates executing code and caches it for subsequent execution. Policy enforcement in Strata will likely be similar also, only this policy will have the ability to change based on the current state of the system. The adaptive policy will be based on a hybrid of current policies, such as the Chinese wall policy, audit policies, and information

flow policies [8], or may utilize the ability to completely switch between the policies as the situation permits.

5. Statement of Project Activities

From initial background research to completion of the technical report, this project can be divided into several discreet steps. Most background research has been completed in order present this proposal. The structure of the project and the resources required to complete it are described in the following sections. A tentative schedule is also presented. The project activities section exhibits plans for carrying out the implementation and data collection portions of the project. Conceptually, it begins immediately following thesis definition.

Activities

1. *Determine a set of arbitrary but representative security policies to mitigate the adaptation of a global security policy.* Note that there may be some question as to the appropriate scope of this project. The first goal is to prove that enforcing an adaptable security policy using SDT is possible and practical for implementation and distribution. This only requires the policy to be arbitrary, with the resulting emphasis placed only on the adaptive behavior of the finished product. Initially, the model will not be required to additionally support advanced computer security practices. This initial step, however, is trivial considering that the flexibility of SDT systems greatly increases the probability of successful results. To ease the burden of future research, the initial set of policies will be, on some level of abstraction, representative of useful computer security practices. This initial

distinction will especially take into account a policy's potential to foster dynamic behavior. A policy predicated on the static analysis of a process or the use of a pre-existing database would not be a likely candidate to fully exploit the capabilities of SDT systems.

2. *Determine the stimuli that will invoke changes, or adaptations, in the policy to complete an initial working model of an adaptive security policy.* Recall that a primary motivation behind this research is that enforcing policies is computationally expensive, so ideally action should only be taken when a security threat is fully justified. These justifications must be expressed rigorously and with a degree of certainty necessary to ensure the consistent recognition of these threats. Note that these justifications are dependent on the type of policy being enforced.

3. *Implement this first working model into Strata.* This step will consist of working with Kevin Scott (see Personnel) a great deal. The simplicity of the first policy model will be crucial in dealing with any implementation issues in Strata that arise due increasing complexity of the model. If the complexity of implementation turns out to have been underestimated, tradeoffs may be made with the simplicity of the initial model. The research will by no means be lost, as it will be incorporated into the following steps. An initial implementation will facilitate the following steps.

4. *Assess the results of this model with proper security and performance testing.* The first goal of performance testing will be to observe the desired adaptive behavior where in response to certain pre-defined stimuli, the security policy

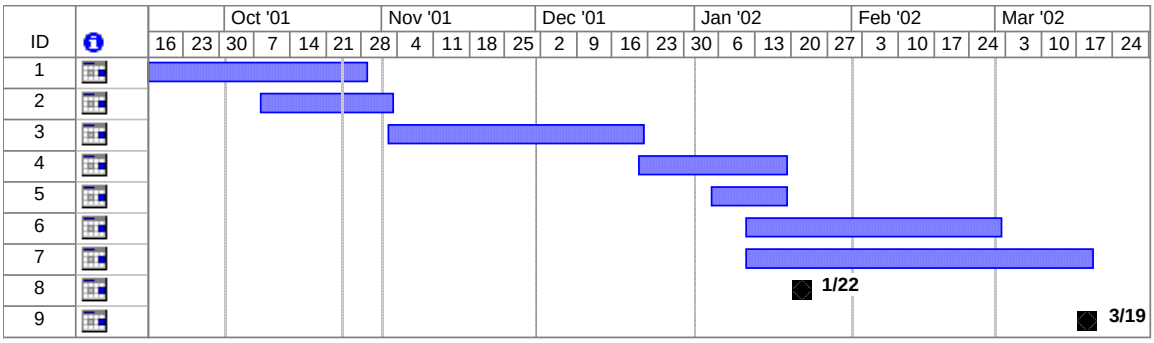
being enforced on a process will be altered dynamically. In particular, a process will run some sequence of events, denoted by *k*, followed by a single event or another sequence of events that justifies action by a policy. A response to this action will either be derived from the current policy (which could be that no policy is enforced) or to spawn a new policy with a different level of intrusion sensitivity. In the latter case, the same sequence *k* will be run, so that the same code can be observed in different situations. This will give a clear indication of the computational savings incurred by selectively enforcing the more sensitive policy during execution as opposed to doing so the entire duration. As for the first case, testing in this area will be indicative of the correctness of the policies themselves. This step could potentially be ignored if the policies are actually arbitrary, and testing for security improvements would be somewhat futile. Actions taken by the polices will be observed in later testing, when they become more representative of current security practices. In order to improve in this area, it will require testing not unlike any other computer security system; real-world testing.

5. *If these results are acceptable, begin to construct an adaptable security policy that may be useful to the computer security community by improving the current policies.* This final step will be iterative and will continue as time permits. Any improvements in the policies will be verified and documented.

Schedule

The final version of the technical report must be completed by the third week of March 2002. This implies a tentative schedule, that the research must be

completed leaving ample time to prepare the report. Refer to the Gantt chart below, Figure 2., for an initial schedule of this thesis project. Note that the schedule is constructed to allow for more research to be done during the semester break. A major portion of this project will be implementing the first iteration of the adaptable security policy. Testing and verification will follow, along with the iterative process of improving the policies. Based on the advice of my technical and security advisors, a desirable stopping point for this process will be determined and the preparation of the technical report will begin.



1.	Background Research
2.	Design First Policy Model
3.	Implement First Model
4.	Test First Model
5.	Verification and Validation of First Model
6.	Explore and Implement more efficient adaptive policies
7.	Prepare Technical Report
8.	Progress Report Due
9.	Technical Report Due

Figure 2. Gantt Chart: Tentative Schedule for Project

Personnel

1. **Author of Technical Report:** *Undergraduate Student Paul H. Lamanna*

   As a forth-year undergraduate student at the University of Virginia, enrolled in the computer science curriculum in the School of Engineering and Applied Science, I will be the primary author of the technical report of the thesis project proposed. I attribute my qualifications for performing the proposed thesis to past experience (see A2.) as well as an avid interest in computer security and a reverence for consumer privacy.

2. **Technical Advisor:** *Assistant Professor Kevin Skadron*

   Kevin Skadron is currently the director of the Laboratory for Computer Architecture at the University of Virginia (LAVA). He specializes in computer architecture and simulation methodology and will serve as my primary source of aid in structuring the project process. We will meet once per week, in person or via e-mail to discuss progress.

3. **TCC Advisor:** *Professor I.H. Townsend*

   I.H. Townsend has been directing senior theses at the University of Virginia for 28 years, and will help me with all the thesis related documents and oral presentations, making sure that I understand the impact of my project, as well as communicate clearly what it is and why I am doing it to both an interdisciplinary as well as a technical audience.

4. **Computer Security Advisor:** *Assistant Professor David Evans*

   David Evans is currently teaching an undergraduate computer security

class and is also advising several security related projects including my own. He will provide valuable advice concerning the security policies to be enforced, specifically how they should be defined and then enforced.

5. **SDT/Strata Advisor:** *PhD Student Kevin Scott*

   Kevin Scott is the author of the SDT framework, Strata. He will provide invaluable assistance through out the implementation of the proposed application.

6. **Additional Advisor:** *Teaching Assistant Professor Christopher Milner*

   Christopher Milner instructed me in an undergraduate computer architecture course. His knowledge of various architectures and also hacking techniques will provide valuable background information for this project.

Resources

1. **Computational Facilities**

   Access to the servers in the computer science building (Olsson Hall) will be required to perform the necessary research. Due to the hazard of encountering malicious code, an isolated machine will be required to ensure the safety of the Olsson Hall network. The Computer Science Systems Staff will provide these services.

2. **Strata/Related Software**

   Strata is an SDT implementation, developed at the Laboratory for Computer Architecture at the University of Virginia (LAVA) by Kevin James Scott and Professor Jack Davidson. Strata, as well as updates and future releases,

will be provided by the authors. Related software is available under the GNU license or can be obtained via my technical advisor.

3. **Research Materials**

   All research materials will be provided through the University of Virginia Libraries, ACM, IEEE, NECI Research Index, and the suggestion of knowledgeable professors.

6. Expected Outcomes

Previous research in SDT has been very successful in integrating various applications into its infrastructure, but the forum usually then turns to performance issues. My research is certainly not exempt from these kinds of concerns since the propagation of security applications greatly depends on their degree of transparency and utility. Affecting computational performance would hinder the practicability of this tool, as well as its future development. But, I am fully confident that performance issues will be kept to a reasonable level in this research, and possibly be improved in later research.

As for verifying that adaptable security polices are a viable application for SDT, I have no doubt that this research will prove that concept. A representative set of security policies will also aid in the feasibility of the research. Later research may then accept the challenge of exploring even more representative sets of adaptive policies to support.

7. Bibliography

Software Dynamic Translation

1. V. Bala, S. Banerjia, and E. Duesterwald. *Dynamo: a transparent dynamic optimization system*. In SIGPLAN '00 Conference on Programmng Language Design and Implementation, pages 1-12, 2000.

2. Ebcioglu K., and Altman, E.R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In Proceedings of the 24th Annual International Symposium on Computer Architecture. 26-37.

3. Kevin Scott and Jack Davidson. Strata: A Software Dynamic Translation Infrastructure. Technical Report CS-2001-17, Department of Computer Science, University of Virginia, July 2001.

4. Kevin Scott, Jack Davidson, and Kevin Skadron. Low-Overhead Software Dynamic Translation. Technical Report CS-2001-18, Department of Computer Science, University of Virginia, July 2001.

Security Policy Enforcement

5. Tim Fraser, Lee Badger, and Mark Feldman. *Hardening COTS Software with Generic Software Wrappers*. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 1999.

6. David Evans and Andrew Twyman. *Flexible policy-directed code safety*. In IEEE Security and Privacy, Oakland, May 1999.

7. Ulfar Erlingsson and Fred B. Schneider. *SASI enforcement of security policies: A retrospective*. In Proceedings of the 1999 New Security Paradigms Workshop, Caledon Hills, September 1999.

8. P. Devanbu and S. Stubblebine. *Software engineering for security: A roadmap*. In A. Finkelstein, editor, The Future of Software Engineering. ACM Press, New York, 2000.

9. Laurence Cholvy and Frederic Cuppens, "*Analyzing Consistency of Security Policies*," in Proceedings of the 1997 IEEE Symposium on Security and Privacy, pp. 103-112, IEEE Computer Society Press, Los Alamitos, CA, 1997.

10. F. Cuppens and C. Saurel. *Specifying a Security Policy: A Case Study*. In Proc. of the computer security foundations workshop, Kenmare, Co. Kerry, Ireland, 1996.

Intrusion Detection

11. Ghosh, A.K., A. Schwartzbard, M. Schatz, *Learning Program Behavior Profiles for Intrusion Detection*", Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring, April 9-12, 1999, Santa Clara, CA.

12. K Jain and R Sekar. *User-level infrastructure for system call int erposition: A platform for intrusion detection and confinement*. In ISOC Network and Distributed System Security, 2000.

13. C. Warrender, S. Forrest, and B. Pearlmutter. *Detecting intrusions using system calls: alternative data models*. In Proceedings of the 1999 IEEE Symposium on Security and Privacy, pages 133--145. IEEE Computer Society, 1999.

Anti-Hacking

14. Smith, Nathan P. *Stack Smashing Vulnerabilities in the UNIX Operating System [online*]. Available WWW: <http://reality.sgi.com/nate/machines/security/nate-buffer.ps> (1997).

15. Tsai, Timothy and Navjot Singh. *Libsafe: Protecting Critical Elements of Stacks*. White Paper 3-21-01, Avaya Labs, Avaya Inc. February, 2001.

16. Hashii, Manoj Lal, Raju Pandey, and Steven Samorodin. *Securing Systems Against External Programs*. IEEE Internet Computing, 2(6):35-- 45, Nov-Dec 1998. 13

17. Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

18. David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In Proceedings of the 2001 USENIX Security Symposium, 2001.

Code Safety

19. Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. *Efficient Software -Based Fault Isolation*. In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 203--216, December 1993.

20. Dexter Kozen. *Efficient code certification*. Technical Report 98-1661, Cornell University, Department of Computer Science, 1998. Available from http://www.cs.cornell.edu/kozen/secure.

21. George Necula. *Proof-carrying code*. In Conference Record of the ACM Symposium on Principles of Programming Languages. ACM Press, January 1996.

Computer Architecture

22. System V Application Binary Interface, SPARC Processor Supplement, Third Edition.

8. Appendices

A1. Budget and Equipment Checklist

All equipment will be provided by the University of Virginia Department of Computer Science. Therefore, the current the budget for this thesis project is minor, requiring only computational resources and SDT and computer security related software. All of these resources are readily available and will require no outside funding.

A2. Biographical Sketch of Student

I am forth-year undergraduate student at the University of Virginia, enrolled in the computer science curriculum in the School of Engineering and Applied Science. I have spent the summer months of 2001 researching possible applications for Software Dynamic Translation, and becoming familiar with the Strata source code. I also became familiar with hacking techniques and a multitude of security related products. I attribute my qualifications for performing the proposed thesis to this experience as well as an avid interest in computer security and a reverence for consumer privacy.

A3. Preliminary Outline of Technical Report

Frontispiece
Title Page
Forward

Table of Contents

1. **Glossary of Terms**
   All terms included will be written in italics throughout the technical report. The glossary will include sections relating to security terms, SDT terms, etc…
2. **Abstract**
   The Abstract will more than likely be identical to the one proposed.
3. **Introduction**
   a. Thesis of Thesis, what, how, and why.
      Adaptive policies using Strata. Dynamic Behavior
   b. Problem Definition
      i. Context
         Current state of security, briefly explain current static practices.
      ii. Concepts
         SDT and Strata, Security policies vs. hacking
4. **Literature Review**
   Similar to proposal, more detail on current state of security, explain many existing applications and their shortcomings. Intro SDT and Security policy enforcement.
5. **Rationale**
   Similar to proposal, advantages with adaptive SDT, goals of tool, flexibility, transparency…
6. **Overview of Contents**

Text Body of Thesis

- Architecture of Strata
- Modifications for fast-strata and secure-strata
- Security Policy Enforcement API Description
- Four added Functions

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

- Examples of Basic Policies.
- Description of Adaptive Policy Behavior
- Display of Implemented Adaptive Policies and Functionality
- Discussion of Thesis goals
- Performance issues with Adaptive Policies
- Related Work

Conclusions

1. Summary
2. Interpretation
3. Recommendations

Bibliography
Appendix