A SECURITY ARCHITECTURE FOR SURVIVABLE SYSTEMS

Chenxi Wang, John C. Knight

Department of Computer Science
University of Virginia

151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740

cw2et@cs.virginia.edu *knight@cs.virginia.edu*

(804) 982-2216

Abstract

The protection of survivability mechanisms against security attacks is a difficult but extremely important problem. If this mechanism were penetrated in any particular system, the adversary might gain control of the entire associated information system. What is needed is a mechanism to preserve the execution integrity of software despite its untrustworthy execution environment. In this paper, we present a comprehensive strategy for protecting survivability mechanisms against attack by adversaries with access to significant resources. The approach uses a variety of forms of diversity at the system level and a general strategy for defeating static analysis at the local level. We refer to the approach to defeating static analysis as One-Way translation and describe the concepts, the underlying theory, the performance, and the implementation.

A SECURITY ARCHITECTURE FOR SURVIVABLE SYSTEMS

1 INTRODUCTION

Today's society is heavily dependent on a number of large information systems that constitute the foundation of everyday activities. The management of transportation systems such as freight rail and air traffic control, the operation of telecommunications systems, nationwide control of electric power generation and distribution, and the basic operation of the financial system are but a few examples of this class of information system. Such systems are often referred to as *critical infrastructure systems*.

Societal dependence on these systems is growing and will continue to grow for the foreseeable future. This has raised concerns about the dependability and survivability of these infrastructure systems [PCCI97, DSB96]. Recent experience has shown that these systems fail for many reasons including hardware failure, software failure, operator errors, and malicious attacks [KEP97, RISK94].

Researchers have proposed a number of approaches intended to improve the survivability of critical infrastructure systems, many of which employ network-wide architectural support to enhance system survivability. These schemes often rely on a distributed monitoring mechanism in which the system in its entirety is monitored in order to detect undesirable state changes including benign faults, malicious behaviors, and other anomalies [PONE97, GRID97, IDIP97, NEUM00]. A number of systems also deploy automated response mechanisms in which real-time changes to the system are determined based on the monitoring information, and the system is reconfigured to reflect the changes ([IDIP97, SKDG98]).

Unfortunately, the introduction of network-wide monitoring-and-control elements can lead to new vulnerabilities. For example, if the control elements were to fail, the failure might result in a system-wide loss of services. Far more significant, however, is the fact that these elements can themselves be the target of malicious attacks. If an intruder were able to corrupt the monitors, he or she could send fraudulent state information to the controllers to cause erroneous state changes. Moreover, if the control elements are penetrated, the intruder would gain access to the control of the entire network simply by manipulating the control mechanism. This is in contrast to a typical network intrusion in which an attacker might be able to compromise a single node (albeit perhaps an important one) but where access beyond there usually requires additional knowledge and resources.

Protection of the network-wide survivability elements is, therefore, essential to survivable systems. In part, traditional security mechanisms can be employed. For example, some components of the control mechanism can be executed on dedicated hosts that are physically isolated and protected, and cryptographic techniques could be used to prevent tampering of communications between these control hosts and the rest of the system. However, parts of the control mechanism need to reside on the monitored hosts to collect data and perform local reconfiguration tasks, and it is the protection of these components that is the most difficult: it must be assumed that the monitored hosts are vulnerable to security attacks (hence, in part, the need for security monitoring). We must allow the possibility that any software running on top of the monitored hosts can be compromised, and that some adversary will try to impersonate or tamper with the program to perform malicious tasks.

What is needed here is a mechanism to preserve the execution integrity of software despite its untrustworthy execution environment. This is exactly the motivation for this research. This document contains a comprehensive, high-level description of the problem and the devised solution to date. As such, this paper should be utilized mainly as a documentation of creative thoughts and a roadmap of the research.

The rest of the document is organized as follows: Section 2 presents the problem context, assumptions and scope of this research. Section 3 describes the fundamental observations that constitute the underlying principles of our approach. Section 4 presents the solution framework and a few examples of the techniques. Section 5 analyzes the security strength of the approach. Section 6 presents related work, and Section 7 concludes the paper by presenting future research directions.

2 THE SYSTEM CONTEXT AND ASSUMPTIONS

In this section, we describe the system context and the assumptions based on which this research is established. We first review the relevant characteristics of critical infrastructure systems and the survivability architecture, as these characteristics provide the necessary backdrop for the solution approach. We then present a set of assumptions to set the problem context.

2.1 Critical Infrastructure Systems

Critical infrastructure systems are complex by nature. Listed below are some characteristics of infrastructure systems that are important to this research.

- **Large scale:** Critical infrastructure systems often involve a large number of computing nodes that are geographically dispersed. These computing nodes and their interconnections provide the computation, data storage, and communication that are needed to provide services. The exact topology of the network is not important, although such networks are usually not fully connected. A typical infrastructure system uses point-to-point links between computing nodes, and the nodes can communicate with one another in any fashion that is desired by the applications.
- **Heterogeneity:** Another defining characteristic of critical infrastructure systems is the degree of heterogeneity. First, the infrastructure system is often composed of subsystems with diverse operational policies and environments—consider regional banks within the same banking infrastructure—and a wide variety of software and hardware components. From a security standpoint, this degree of heterogeneity implies incongruity in the security policies and mechanisms employed; some sites will be more easily penetrated than others. A universal protection mechanism such as securing each individual host from the ground up is both impractical and infeasible.

  Second, critical applications usually do not consist of a set of similar programs (such as mail servers) cooperating to achieve some goal. Rather, the programs running on different hosts often serve distinct purposes, and they must cooperate in some form of sequential processing in order to provide desired services. A direct consequence of this diversity is that functionality is not uniformly distributed across the system; some computing nodes provide services that are more important than the services provided by others. In order to make system-level management decisions, it is therefore necessary to correlate and integrate local information. This implies complexity in enforcing survivability as well as inherent difficulties in securing the survivability mechanism.

- **Legacy and COTS components:** The software employed in infrastructure systems tends to be large and to make extensive use of both legacy and Commercial-Off-The-Shelf (COTS) components. This means that any mechanism we introduce must consider the uncertainty of COTS and legacy software in terms of their reliability and security. Moreover, the characteristics of the operational environment are determined largely by the applications—retrofitting the system with survivability mechanisms is particularly difficult for one cannot mandate drastic changes to existing system architectures (such as demanding changes to the network topology) or system and application software (such as demanding that the applications be rewritten).

2.2 The Survivability Architecture

It is in the context just described that survivability mechanisms have been proposed. An example mechanism is the hierarchic, adaptive, control-system architecture developed here at the University of Virginia [SKDG98]. The research henceforth is carried out in the context of this architecture. We briefly describe this survivability architecture below.

2.2.1 A Control System Construct

A critical component in the survivability architecture is the *control system* construct. Introduced as an external entity to manage the infrastructure system, the control system operates in parallel with the infrastructure system. Its primary function is to monitor the system operation, and make management decisions to enforce system survivability. For example, the control system may initiate dynamic system reconfigurations in order to continue services in the event of failures or security attacks.

The control system consists of a set of control servers and a collection of probing and actuating programs that execute on infrastructure hosts. The probe programs collect information about the critical application and send that to the control servers to be analyzed. The actuators accept reconfiguration commands from the control servers and initiate local changes that are implied by the control analysis. An abstracted view of such a control system architecture is shown in Figure 1.

The control hosts (represented as gray circles) are organized in a hierarchical structure with control actions being carried out in a number of different levels. They interact with the controlled system through a set of sensors and actuators (represented as black dots), which execute on the infrastructure hosts (represented as white circles). The sensors and actuators communicate monitoring information to the control servers and perform reconfiguration commands issued from the control servers. Control hosts are interconnected as appropriate.

Note the control servers are physically separate from the rest of the system. There are several advantages to such a design. First, executing control algorithms locally on application hosts can be a significant resource drain—running them exclusively on the control servers is beneficial for efficiency reasons. Second, there tend to be fewer numbers of control servers than there are application hosts, therefore it is possible to have dedicated control servers, and to secure and configure them independently. This implies a rigorously controlled execution environment for the control servers, and consequently, enhanced security and assurance.

2.2.2 The Probing and Actuating Mechanism

The primary function of the probes is to collect system state information as input and generate sensing values that are sent to the control servers for processing. Communications between the probes and the control servers follow a prescribed protocol that includes a timing mechanism (e.g. time-outs), predetermined data format, and designated hand-shaking sequences. Details of this communication will be discussed in later sections. It is important to note that the sensor program communicates with the control servers in a prescribed fashion, and adhering to the interaction protocol is considered expected behavior of the sensor program.

If necessary, the control servers make response decisions and communicate them back to the actuators to take effect. The actuator process accepts the control command and undertakes the actions needed locally to reconfigure the application. Examples of such dynamic reconfigurations include shutting down communication lines, dynamic process migration, active load balancing, etc. Interested readers can find details of this actuating process in other reports [ELKN99].

The probing and actuating functionality need to reside on the monitored host to take effect. They can be implemented as either separate processes or as distinct functionality within a single process. In the rest of this work, we will refer to the probe and actuator on the same host as a collective entity—the monitor process.

2.3 Security Issues

The survivability architecture, as described above, gives rise to a series of security concerns. In a

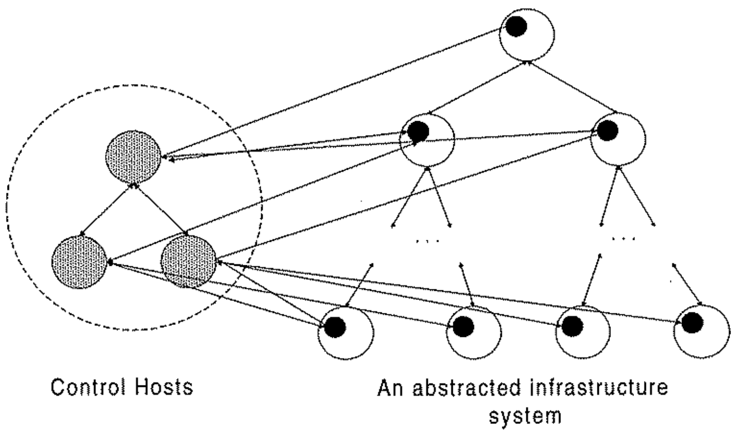


Figure 1. Example Controlled Infrastructure System

nutshell, these security concerns can be grouped into three loosely-defined categories.

- **Protection of the control servers.** The control servers execute the control algorithms, and therefore are at the heart of the whole control mechanism. The servers and the software executing on top of them must be protected from corruption and denial-of-service attacks.
- **Protection of communication.** Monitoring and control communications occur over the network. This network traffic needs to be authenticated, and protected against tampering, eavesdropping, insertion and deletion attacks.
- **Protection of the monitors.** The monitor process needs to be protected so that the data gathering and the actuating part of the mechanism can be trusted.

These protection issues are closely related; that is, techniques used to secure one part of the mechanism will impact the protection of others. For example, cryptographic methods are often used to protect network communication. However, cryptography is not sufficient if the communicating hosts (and consequently the secrecy of the cryptographic keys), are not adequately protected.

Because of the multitude of security issues, complete protection of the survivability architecture in its entirety, i.e., against every possible combination of attack scenarios, is beyond the immediate scope of this work. To simplify the problem and to help focus the task of this research work, we make the following assumptions:

- **Trustworthy control servers.** The control servers, physically separated from the rest of the system, are *dedicated* to performing the algorithmic part of the control mechanism. Securing the control servers requires a combination of careful system administration (e.g. exercise strict control on what software is allowed to execute on these servers) and good physical security (e.g. restricted access). For the purpose of this work, we assume that the control servers are secure and trustworthy; that is, the possibility of a successful security attack on the control servers is considered sufficiently low such that the software running on top of them can be trusted to perform its task correctly.
- **Secure communications.** We assume that all network communications occur over authenticated channels equipped with cryptographic protections, and that the cryptographic techniques are sufficient, in this context, to protect the freshness, integrity and privacy of the communication. It is important to note that the use of cryptography does not prevent denial-of-service attacks—network traffic can be deleted and communication channels can be jammed. An exception is the inter-server communication among control servers. The control servers communicate with one another over dedicated links with restricted access, and these links are assumed secure against eavesdrop, traffic insertion and deletion attacks.
- **Un-enhanced application hosts.** The infrastructure system is a network enterprise with tens of thousands of computing hosts. It is virtually impossible to enhance every application host to be rid of security concerns. We hence assume that application hosts are running in their normal operational mode—no special enhancement other than the security mechanisms that are already in place for the applications. This assumption has many implications. In particular, it implies that the application software and infrastructure hosts might contain security flaws, and that they might be vulnerable to a variety of security attacks.

- **Trusted survivability software.** Many real-world security problems arise not because of flaws or oversight in the design of the security mechanism, but rather they are caused by errors in the software implementation. It is not our objective, in this work, to address security flaws of the latter kind. Throughout this work, we assume that the survivability software is trusted in such a way that it operates as expected (if not compromised), and it does not contain any *malicious* flaws that will lead to a compromise of the survivability mechanism. This assumption allows us to focus on security threats from the environment and protection issues that are not implementation specific.

These assumptions state that, in the final analysis, the weak link lies in the protection of the probing and actuating mechanism on untrustworthy application hosts. Furthermore, we are primarily interested in sophisticated attacks that fall in the category of *intelligent tampering* and *impersonation* attacks:

- **Intelligent Tampering**. Intelligent tampering refers to scenarios in which an adversary modifies the program or data in some specific way that allows the program to continue to operate in a seemingly unaffected manner (from the trusted server's point of view), but on corrupted state or data. Overwriting data buffers with data of the correct format but different values is an example of such an attack. Under this definition, tampering with the software in a random way (e.g., overwriting random bits in the memory) does not constitute an intelligent tampering attack.
- **Impersonation.** An impersonation attack is accomplished when an intruder successfully emulates the observable behavior of the legitimate program. Similar to intelligent tampering, impersonation attacks seek to establish a rogue version of the legitimate program. The difference lies in that tampering deals with internal specifics of the program while impersonation operates at the level of observable semantics.

Both intelligent tampering and impersonation require some level of understanding about the target program—whether it is knowledge about the external behavior or the internal specifics of the program, some information is needed a priori for such an attack to succeed. Consider a scenario in which the intruder's objective is to forge monitor messages to the control server, and assume all messages are signed with the monitor's private key. The knowledge the intruder must acquire, among others, is the private key of the monitor program, for message forging is not possible without the correct signature key.

The ways in which an intruder can obtain the knowledge necessary to compromise the program depend heavily on the intruder's capabilities and resources. These capabilities and resources also have a significant impact on the design of the defense mechanism. Further assumptions about the intruder's capabilities and the attack scenarios have to be made before any analysis can be undertaken.

Three categories of intruders, classified by their respective capabilities, are likely to be present in the context of this work. Listed in the order of increasing level of capability, they are: *Network Intruders, Malicious Insiders,* and *Privileged Users*. This classification is based on the one used by Aucsmith [AUCS96] in his Integrity Verification Kernel (IVK) work. This section discusses each category within the context of the survivability architecture described earlier.

Network Intruders: This category refers to intruders who do not have direct access to the host where the monitor program executes. These intruders access the system through network entry points, and they can eavesdrop on the communication line, and insert and delete network traffic.

A typical network intruder is bounded by communications protocols and other network-based security mechanisms (e.g. firewalls, network access control, etc.). Their mission is to either breach the host security perimeter (i.e. getting in from outside) or become a *man-in-the-middle* by forging or replaying communication messages. This type of intrusion is best dealt with by the deployment of correctly designed and implemented security protocols and proper administration, and therefore is not covered by this work.

It should be noted that network intruders can gain further access by exploiting flaws in communication protocols or network security mechanisms. For example, a successful buffer overrun attack may render more privileges to the intruder as a result. In doing so, intruders from the network may become malicious insiders or privileged users who could possess significantly more powerful capabilities than a typical network intruder.

Malicious Insiders: This category refers to adversaries that have control of some program running on the targeted host. These intruders can be legitimate users of the system, or an outsider who has gained illegal access to the host system.

Malicious insiders have access to some system resource, and they can manipulate the programs under their control or introduce Trojan-horse programs to inflict damage to other applications or the underlying host system. An example of such a malicious insider is someone who has obtained the password of other users and is now able to read and write private data and programs that belong to the compromised accounts. In general, actions of malicious insiders do not directly undermine the security of the host system (e.g. they generally do not compromise the operating system). For the purpose of this discussion, It is assumed that these adversaries are still bounded by the operating system and its security mechanisms.

Malicious insiders are intruders without the "root privilege". Actions of a malicious insider can be greatly limited by the use of properly designed access control mechanisms, competent intrusion detection tools and careful administration. At best, intruders in this category can cause denial of services or instantiate malicious software such as virus or Trojan-horse programs to effect damage to the target program. However, they usually do not employ sophisticated program analysis tools, and hence are not candidates for intelligent tampering and impersonation attacks. To defeat malicious insiders, the approach described here relies on the principle of diversity to reduce software uniformity, which is often the cause for successful virus or Trojan-horse attacks [AUCS96, FOSO96].

Privileged Users: Adversaries in this category have direct access to the host where the target program is running. In addition, they may have the following privileges.

- Access to private memory of other user or system processes
- Access to source code of the target program
- The ability to introduce and execute random software on the host

- The ability to manipulate and replace system software

Read access to the host memory implies that the adversary can obtain the binary image of a loaded executable program. That includes code as well as data associated with it. Write access gives the perpetrator the ability to modify main memory.

Access to source code of the target program suggests that software protection should not, and cannot hinge on the obscurity of the source program—determined adversaries will acquire copies of the original source program via out-of-bound means, and it is virtually impossible to guarantee such secrecy. However, a compromised source program does not necessarily imply direct compromise or immediate knowledge of the running executable program. That is, an executable program generated from a known source program, aside from being functionally equivalent, could contain extensive syntactic or semantics differences from the source program such that impersonation or intelligent tampering of the binary would still require analysis of the executable program. This premise is the cornerstone of much of this work, and we will elaborate in later sections why the premise stands and how it can be exploited as a basis to devise software protection mechanisms.

The ability to introduce and execute random software implies that the intruder may have access to specialized software analysis tools such as debuggers, decompilers and system diagnostic utilities. They can perform analysis on-line such as system diagnostics, or off-line such as black-box testing, execution emulation, and break-point-based debugging.

The ability to manipulate and replace system software suggests that the host security mechanisms such as those provided by the operating system can be compromised. Therefore any mechanism deployed to protect the software should not depend on the authenticity or security of the host OS. This assumption is, of course, the most troublesome—once you allow the host OS to be compromised, the perpetrator may have near complete control of the platform, and their action henceforth is only limited by available resources.

There is, however, one restriction on the intruder capabilities—he or she may not substitute or install hardware on the host system. Altering hardware configurations requires physical access to the host system. It is reasonable to assume that such access is discouraged, or to a large extent, difficult to obtain. This assumption eliminates the possibility of special hardware analysis tools such as bus analyzers or hardware monitors deployed directly on the target host. It should be noted that the use of external hardware for off-line analysis is still a possibility.

At this level of sophistication, the adversary has access to ample system resource and a great deal of knowledge on how the system works. Security attacks from these adversaries are the most powerful and also the most difficult to defeat. In fact, no security mechanism exists and none could be developed that will provide protection against such adversaries in the absolute sense—there may not be any workable solution against perpetrators with unbounded resources. What we aim to do in this work is to:

- Increase the technical difficulty to deter security attacks from malicious insiders and privileged users, and
- Understand and provide a theoretical basis to determine exactly what returns each protection

mechanism provides in order to make informed decisions.

3 Exploring the Solution Space

In this section, we explore the solution space for the problem described in the previous sections. A realistic solution for the software protection problem is to raise the technical bar for launching a feasible attack, i.e., the majority of the impersonation or tampering attacks must be either computationally or economically infeasible to accomplish. With this goal in mind, we explore a complexity-driven solution space.

The basis for judging technical difficulty is the complexity of computation involved in the operation. The notion of *computational complexity* here is used in a slightly unconventional sense. For example, computational complexity, in the traditional sense, is dominated by the order of growth of the algorithm—an operation that is of the order $O(n)$ is considered more efficient and less complex than one that is of $O(n^2)$. This is regardless of the lower order terms of the running-time formula and the constant coefficient of the leading term that determines the order of growth. In this work, it is not the order of growth alone that is of interest. Instead, what of interest is the *operational complexity,* affected by the input size, the constant coefficients and the order of growth. For instance, if the complexity of an attack algorithm can be expressed as

$$an^x + bn^{x-1} + \ldots + c\,,$$

with n being the input size, a feasible defense mechanism might aim to raise the order x, the input size n, or the most significant coefficient a.

The discussion in section 2 states that intelligent tampering and impersonation attacks require certain information about the program. This information is the identification secret (or secrets) that constitutes the basis of the program's identity (without this secret, the trusted server cannot differentiate between a legitimate program and an imposter). The identification secret can appear in the form of protocols, cryptographic keys, or a particular set of behavior patterns that can be verified by the trusted server. For the purpose of this discussion, we assume that the target program carries such identification secrets, and that there exists a mechanism for a trusted server to authenticate the secrets and verify to whom it is communicating.

The presence of such an authentication mechanism implies that impersonation or tampering is impossible without performing program analysis to determine what the secret is, and how it will be verified. For example, consider the following code sequence:

```
int a = function1( );
int b = function2( );
Check_for_intrusion(&a, &b);
...
p     = &a;
...
Integrity_check(p);
```

If an adversary were to tamper with the `Check_for_intrusion()` function, they need to understand whether and how the `Check_for_intrusion()` function changes the values of `a` and `b`, and whether the value of `a` and `b` are used later in the program. Without this knowledge, the adversary's action might be revealed when `Integrity_check(p)` is called.

In general, an adversary aiming to tamper with the program in an intelligent way must understand the effect of his actions, and this boils down to an understanding of the program semantics. One way this understanding can be acquired is through program analysis, and thus the operational complexity of intelligent attacks is determined principally by the complexity of program analysis. The solution framework used throughout this work incorporates various techniques to increase the complexity of program analysis, and thereby decrease the economic incentive of the attack.

Before discussing techniques to obstruct program analysis, we present a model of complexity for program analysis. The purpose of program analysis is to obtain information, and the complexity model is therefore established based on the information being analyzed. The model has three dimensions:

- **The amount of information:** Intuitively, program analysis is more complex the more information that must be analyzed. For example, attacking the network-wide survivability mechanism requires the compromise of a collection of monitor programs at different locations. If the monitors are diverse, independent analysis efforts will have to be expended to analyze each monitor. More effort (hence complexity) is required than would be required if all monitors were identical.
- **The computation of analyzing the information items:** There is a cost in complexity in analyzing each information item. This computational complexity is a significant factor in the overall difficulty of the program analysis.
- **The information lifetime**: Operational complexity, when rated against available resources, can be measured in terms of time. Increasing the amount of information and the complexity of analyzing the information can be viewed as efforts to increase the time required for an attack, while limiting the lifetime of the information serves as a complimentary tactic—it imposes a time bound within which the attack must be completed. The shorter the information lifetime, the more resources are required for the analysis, and hence the more difficult the attack.

These dimensions determine the complexity of program analysis. Using this model, techniques accentuating one or more of the dimensions present increased difficulties in program analysis. The solution framework used throughout this work is based on this complexity model, and is comprised of techniques that yield varying degrees of complexity along the different dimensions. The components of the solution framework include the following elements:

- **Diversity:** The notions of temporal, spatial, data, and design diversity are explored to introduce complexity and variations in the program. The diversity techniques present means to increase the analysis complexity as well as limit the time window of attacks.
- **One-way Translation:** One-way Translation is a compiling process in which source programs are translated into functionally equivalent but structurally varied binaries. The resulting binary programs incorporate properties that are difficult to analyze. When combined with temporal diversity, this mechanism provides a powerful way to obstruct program analysis. The

translation process is driven by a random number, and therefore is difficult to invert.

- **State Inflation:** This includes a set of mechanisms to obstruct dynamic program analysis techniques such as black-box testing by inflating the program state space.

3.1 Diversity

Diversity is an important engineering principle in building dependable systems. For example, in the design of an aircraft, *geographic* diversity is often used in the layout of hydraulic lines—each of the redundant lines feeding control surfaces pass through different parts of the fuselage and wings. This design helps to ensure dependable operation by tolerating certain perturbations in the environment including various forms of physical damage.

Incorporating diversity into the design of secure systems helps to reduce vulnerabilities that arise from uniform designs that are often the source of replicated flaws [FOSO96]. Four forms of diversity—*spatial, temporal, design*, and *data* diversity—are particularly useful in securing software execution. These forms of diversity have the following meanings in the context of this research:

a) Spatial Diversity: Spatial diversity refers to the placement of different instances of the same software at different locations. In a network, this placement refers to the use of different addresses (in any address space) on different network nodes.

The principles of spatial diversity can be used to thwart class attacks, a type of security attack based on exploitation of the same software or configuration flaws. For example, most script-driven attacks capitalize on a particular set of known flaws, and the same attack may be attempted on thousands of computer systems. If the different program instances are placed at different physical locations throughout the system, an intruder must invest more effort if the goal is to corrupt the network-wide survivability mechanism. Spatial diversity increases the amount of information an intruder must analyze to launch an attack, and is especially important considering that a large number of known security attacks are class attacks [AUCS96, FOSO96].

b) Temporal Diversity: Temporal diversity refers to a periodic variation in the software characteristics over time. Temporal diversity serves as a means to limit the lifetime of information, and hence the time window for a particular attack. If a successful attack takes longer than the lifetime of the information, then clearly it will not succeed.

As an example, suppose that after obtaining the binary image of an executable program *P*, an intruder attempts to perform a systematic state-space search to reverse engineer the binary program. If this effort succeeds after time period ΔT, the result might be a successful tampering or impersonation attack against *P*. However, if the properties of *P* change within ΔT; i.e., if *P* is replaced with *P*', the information obtained at the end of ΔT might prove to be ineffective if used against *P*'.

Temporal diversity implies dynamic changes—a property or a data element may only be valid or have security-related consequences for a limited time. In this work, temporal diversity is realized with periodic replacement and reorganization of the binary program and its properties.

c) Design Diversity: Design diversity is the use of different designs within several programs that implement the same specification. It has been employed in various forms of fault-tolerant software including recovery blocks and N-version programming.

Design diversity holds promise in the security area addressed in this research because of the possibility of detecting tampering in a subset of the versions by observing differences in outputs. Multiple versions of the software to be protected would be written with each one prepared by an independent team. At execution time, all of the versions would be executed in such a way that their outputs could be compared with any deviation from the majority indicating a fault of some sort, possibly tampering.

d) Data Diversity: Data diversity employs multiple copies of the *same* program operating on different data. In a process called data re-expression, the data that the copies use is intentionally transformed in such a way that the output of the software is either identical or almost so for each version of the data.

Data diversity might be employed in protecting trusted sensor software by running multiple copies of the trusted software and checking that each yields the same (or properly related) data for the trusted server. In this way, any tampering would have to affect all versions essentially simultaneously in order to be effective.

3.2 One-way Translation

One of the principal program analysis techniques is static analysis—a technique to analyze program properties by examining the static image of the binary program. Static analysis can reveal program properties such as uses of variables, data locations, etc. This information can then be used in targeted tampering of the program.

A comprehensive static analysis on a program requires, as a minimum, the following information:

- Control-flow information
- Data-flow information

Control-flow information provides knowledge on the program flow control that constitutes the basis of further analyses.

Data-flow information provides knowledge about data quantities in a computer program such as the possible "modification, preservation and usage" of these quantities [HECH77]. Examples of data quantities include variables, instructions and memory locations.

The complexity of static analysis, therefore, depends on the complexity required to acquire the control flow and data flow information. The goal of One-way Translation is to incorporate techniques to obstruct control flow and data flow analysis. Some of the techniques discussed in this research are:

- **Masking control flow:** The control flow of the program can be masked by insertion and restructuring of control constructs. By adding non-functional code and breaking and reorganizing existing control constructs, the program control flow can become arbitrarily complex.

This is designed to obstruct control-flow analysis, a necessary step in decompiling or reverse engineering of programs.

- **Masking code or data content:** Data representations, as well as code constructs, can be restructured in such a way that it will be difficult to recover its original content or even its intent. For example, variables can be divided into subparts, and computations on the variable could be replaced with corresponding computations on the subparts and an operation to construct the correct result. Similar techniques can be applied to arrays, statements and subroutines.

- **Masking code and data location:** Some flow analysis techniques rely on code generation conventions such as the placement of local variables, etc. This type of analysis can be thwarted by the use of random code or data allocation algorithms. Furthermore, certain types of restructuring, when applied to both instructions and data, can also be used to obfuscate code and data locations. For example, a function can be in-lined at its call point or restructured to an arbitrarily different signature.

- **Masking data usage:** One of the primary functions of data flow analysis is to determine the usage of data – when and where they are used in the program. Data usage provides critical information for code optimization, and in this context, it can be used to facilitate intelligent tampering. Data aliases, for example, can complicate the analysis of data usage. Similarly, the use of indirect addressing and pointer manipulation can also be used to mask information on data usage.

These techniques aim to obscure information contained in the program. The premise is that by obfuscating, the resulting program will be more difficult to analyze, thus more difficult to decompile or reverse engineer.

It is important to raise the question of different objectives between conventional program analysis and what is intended in this work. The former has the goal of code improvement, thus a more aggressive, global analysis tactic is desirable. The latter, however, intends to gain specific knowledge to allow targeted program manipulation, and it therefore does not necessarily require as ambitious an analysis strategy. While that may prove to be true in some cases, the ultimate objective of this work is to make the task of program analysis as difficult as possible. In other words, the techniques employed here must include an effort to force the use of the most advanced analysis techniques possible. For example, disseminating critical information throughout the entire body of the program requires a global analysis to gather the necessary information. In so doing static analysis will be ineffective if not used in its most aggressive form, or not applied to the entirety of the program.

3.3 State Inflation—Increase the Complexity of Dynamic Analysis

Dynamic analysis such as black-box testing analyzes the program behavior without delving into the internal details of the program. The goal of this type of analysis is to gain insights into the semantics of the program behavior in order to emulate it. For example, testing the program with different input parameters and observing its output behavior may reveal information that can help determine the program state space.

Dynamic analysis can be conducted via off-line execution and testing. Alternatively, an intruder can attempt black-box analysis by observing the legitimate execution and using Markov analysis-like techniques to infer the relationships between program output and past events in the environment [INGE71].

If the state space of the program regarding input and output is simple, with relatively low effort the intruder can deduce the state space and emulate it to impersonate the legitimate program. For example, consider a monitor program for which there are three basic input states: UP, DOWN, and DEGRADED, and the program outputs an integer 0, 1, and 2, respectively, for each of the input states. In this case, a simple black-box testing would suffice in revealing the entire state space.

To protect against dynamic analysis, we propose the technique of *state inflation*. The purpose of state inflation is to increase the complexity in the state space of the program such that dynamic analysis would provide poor returns on the investment of time and effort. Again, the effectiveness of the scheme is measured in terms of the amount of information an intruder might be able to gather within a prescribed time frame.

The benefit of state inflation is perhaps best illustrated with an example. Consider again the example of a monitor program which operates on three basic input states: UP, DOWN, and DEGRADED. The program generates output integers 0, 1, and 2, respectively, based on the input state. Now consider instead of generating one of the three integers, the monitor applies the following algorithm to generate series of number x's such that:

$$E(k, x1) \bmod 3 = 0$$

$$E(k, x2) \bmod 3 = 1$$

$$E(k, x3) \bmod 3 = 2$$

$E(k, x)$ is a one-way function, and k represents a key that the monitor shares with the trusted server. Instead of transmitting 0, 1, or 2 across the network, the monitor transmits a randomly chosen x's in the appropriate series of numbers (e.g. $x1$ for UP, $x2$ for DOWN, and $x3$ for DEGRADED). The receiver of the integer can then easily compute:

$$E(k, x) \bmod 3$$

to obtain the state information, while an observer, not knowing k, will not be able to determine which x's correspond to which state.

While this example is reminiscent of encryption, what is important here is that the one-to-one mapping between the input states and the output integers are replaced with a one-to-many mapping. Note that there is an arbitrarily large number of output values for each input state, which will appear essentially random to an outside observer.

Dynamic analysis is based on information obtained by observing program execution; that is, each state transition during the program execution disseminates certain amount of information into its environment, and over time the aggregate of this information may be sufficient for an observer to determine the entire state space of the program. What state inflation attempts to do is to expand the program state space and the number of possible state transition for the same operation. Consequently, the average amount of information (i.e. the entropy of information) provided by each state transition will decrease. More effort thus must be expended to gather an equivalent amount of information.

4 ONE-WAY TRANSLATION

In this section we describe a design for a compiler-based mechanism to achieve program diversity and complexity, and in turn to obstruct static analysis of programs. This mechanism, by the name *One-way Translation*, uses compilation techniques to generate binary programs that are resistant to static analyses.

One-way translation transforms a source program into a binary program in such a way that the reverse transformation cannot be determined without the expenditure of tremendous resources. Formally, One-way Translation can be described as follows:

> Let *TR* be the translation process, such that $P \xrightarrow{TR} B$ translates a source program *P* into a binary program B. *TR* is a one-way process if the time taken to reconstruct *P* from *B* is greater than a specific constant *T*.

The core of One-way Translation is the semantically-equivalent transformation of programs to incorporate design diversity and code complexity. When combined with temporal diversity techniques (e.g., a periodic replacement of the program), it provides the necessary elements to deter program analysis, and ultimately defend against intelligent tampering and impersonation attacks.

4.1 A Model of Semantics-Preserving Transformation

We first present a model of semantics-preserving transformation since it departs from the traditional meaning of functional equivalence.

In traditional compiler parlance, *semantics-preserving* transformations preserve the input-output behavior of the program. In other words, the program, before and after the transformation, must produce the exact same results if given the same input [MUCH97].

It should be noted that this traditional definition of functional equivalence is often violated in the actual practice of compilation. For example, the result of commutative operations such as addition, in theory, does not depend on the order of the operation being performed. However, reversing the order of a commutative operation can lead to different results due to possible rounding errors [ref?].

In this work, we employ a relaxed notion of functional equivalence. Function is not defined in terms of input-output relations. Instead, it is defined as a set of high-level specification of tasks. Different implementations, if they fulfill the tasks specified, are considered semantically-equivalent irrespective of their input-output behavior. For example, if a program's function is to report the temperature of the day, two different algorithms—one reporting the temperature in Fahrenheit and the other reporting in Celsius—both accomplish the specified task, and are therefore considered functionally equivalent despite the fact that their input-output behavior is quite different.

Under this definition of functional equivalence, code transformations may affect the internal structure as well as its external observable behavior of a program. Transforms affecting the external behavior of the program may very well alter the program signature. Also note that the traditional definition of functional equivalence is subsumed by the new definition; that is, program transforms that are considered semantics-preserving in the traditional sense are clearly semantics-preserving under the new definition.

The new notion of functional equivalence is application specific—the set of semantics-preserving transformations for one application may or may not preserve functional for others. For example, replacing a DES encryption algorithm with an implementation of RC4 is a functionally equivalent transformation for the purpose of encryption, but would be meaningless where encryption is not concerned. The specification of tasks will have to be derived from the domain knowledge of the application.

The notion that there exists an equivalence class of programs that differ not only in terms of internal representation but also in external behavior is fundamental to the One-way Translation idea. This equivalence class embodies the idea of software design diversity. Recall the discussion of temporal and spatial diversity in the previous sections, and observe that spatial diversity can be achieved by deploying different programs within the same equivalence class at different physical locations, and temporal diversity can be realized by updating the program periodically with copies in the same equivalence class.

4.2 The One-way Translation Compiler

The One-way translation process is based on the idea that the representation of a program, both high-level and intermediate, can be modified to incorporate semantics-preserving transformations. The program modifications are embedded in the compilation process. They constitute the mechanism via which design-diverse programs within the same equivalent class are generated.

The code transformations can be introduced in various stages of the compilation process. However, since one of the goals of transformation is to generate different executable programs, it is advantageous to perform the modification when the memory layout and the instruction stream of the program have not yet been determined. In this work, we choose to perform code transformations at the source and the intermediate representation levels.

It should be noted that functionally-equivalent transformation can also be applied to binary programs directly [CER99, KEHO97]. However, only limited forms of transformation are applicable to binary programs, and the issue of platform-dependence could hinder the wide adoption of the scheme. For these reasons, we choose not to implement code transformation at the binary program level.

The specific transformations applied in each compilation are determined by a random number. Consequently, the One-way Translation process is capable of generating a suite of different programs based on the same original source program. Recovering the original source program from a binary version is "difficult" since it requires knowledge of the randomness in the translation process.

In order to provide the code variation and complexity, two types of modification can be applied to a target program: internal transformations, and behavioral transformations.

4.3 Internal Transformations

This type of transformation affects the internal representation of a program. Such transformations may or may not influence the program's external behavior directly. For example, changing the order of non-interfering instructions alters a binary program's internal structure, but does not affect the result of execution.

Internal transformations can be divided into the following two categories:

- **Control-flow Transformation:** Control-flow transformation modifies an existing program control-flow to that of a more complex form in order to deter static analysis—most static analyses rely on information about the program control-flow to obtain analysis precision.
- **Data-flow Transformation:** The problem of extracting a secret (or secrets) from a binary program can be reduced to the equivalent problem of conducting a data-flow analysis. Data-flow transformations can be applied to complicate the process of analyzing the modification, preservation and usage of data quantities, the basis of most data-flow problems.

4.4 Behavioral Transformations

This type of transformation alters the observable behavior of a program. The following categories of transformation can be applied to the program in order to provide variability in its behavior pattern:

- **Interfacing Protocol Change:** The program interfaces with the trusted servers via a predetermined protocol. This protocol is initialized at each installation to a unique instance. This of course implies changes in both communicating parties—the interfacing program in the trusted server must be changed accordingly.
- **Change of identification secrets:** The program's identification secret such as a cryptographic key can be made installation unique; that is, the one-way translation process generates a unique key for each binary program. At each installation, the program is updated with a functionally-equivalent version with a new key.
- **Change of input-output behavior:** The example in section 3.3 illustrates one scenario in which the input-to-output relation of the program is changed from a one-on-one mapping to a one-to-many mapping. This modification changes the input-output behavior from the outset, but still preserves the meaning of the program.
- **Alternative implementation of function:** This is very much a software component plug-n-play idea. This work relies on user specification as well as alternative implementations supplied by the user to accomplish the plug-n-play model.

In the following sections, we detail the code transformations in the above categories and present a number of example techniques.

5 INTRA-PROCEDURAL CODE TRANSFORMATIONS

Static analysis at the intra-procedural level analyzes information within a particular function. This type of analysis can be classified into two categories: *flow-sensitive* and *flow-insensitive* [BANN79, MARL93]. Flow-sensitive algorithms consider program control-flow information and, in general, yield more precise results than flow-insensitive algorithms. Flow-insensitive algorithms do not make use of control-flow information during the analysis, and therefore must settle with a solution that summarizes over all possible control flow paths. For this reason, flow-insensitive analysis is generally more efficient with the price of being less precise.

It is important to note that control-flow analysis is the first analysis stage—it provides information about the program control transfer that is essential for subsequent data-flow analyses. Without this information, any data-flow analysis is restricted to the basic-block level only and will be fundamentally ineffective for programs where data usage is dependent on program control-flow.

The primary purpose of the intra-procedural code transformations is to conceal the program control-flow and thereby hinder both control-flow and data-flow analysis. The end result of these transformations is as follows:

- Flow-sensitive analysis cannot be more precise than flow-insensitive analysis.
- Flow-insensitive analysis is made ineffective by incorporating data whose usage is control-flow dependent.

In this section, we first present the fundamentals of control-flow analysis and how they are used in this work to conceive anti-static-analysis techniques. We then describe a list of code transformations that are applied in this work.

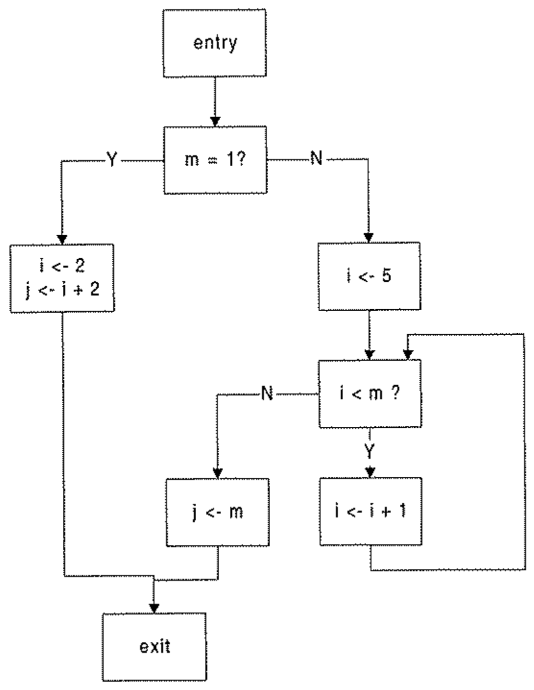


Figure 2. An Example Flowgraph

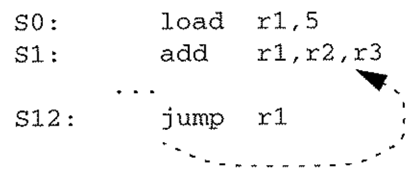
5.1 The Fundamentals of Intra-procedural Control-flow Analysis

Control-flow analysis encodes and makes explicit the flow of control in a program. Intra-procedural control-flow analysis constructs the flow graph for each procedure as follows—consecutive statements within the procedure are partitioned into *basic blocks* such that once the first statement of the block is executed, all statements in the block are executed sequentially. Program control is transferred to another block once every statement in the current block has been executed.

A flow graph represents the flow of control among basic blocks. Formally speaking, a flow graph is a triple $G = (N, A, s)$, such that N is a set of vertices representing basic blocks, A is a set of arcs between blocks, and s is the starting vertex in the graph. An arc(x, y) from node x to node y indicates that program control can transfer potentially from block x to block y. There exists at least one path from the starting node to every other node in the graph. Figure 2 shows an example of a flow graph and the corresponding code segment.

Real-world programs tend to have control-flow that can be easily discerned since this is encouraged for program clarity and enforced by high-level language constructs. In such a program, branch instructions and targets are easily identifiable. Thus determining the program flow graph is a straightforward operation of complexity $O(N)$, where N is the number of basic blocks in the program.

Now consider the case where branch instructions are indirect jumps whose target addresses are not known at compile time. Figure 3 shows such a code segment in pseudo assembly code. In this example, the instruction at `S12` is an indirect branch instruction whose target is defined in register 1. In order to determine to which location this instruction will branch, a static analyzer will have to examine the code to reveal that the content of register 1 is defined at `S1` (the definition at `S1` overwrites the one at `S0`). What just happened here is a *use-and-def* analysis in which a *use* of a variable (whose content held in register 1) is identified and its latest *definition* (at `S1`) is found [MUCH97]. The dashed line in Figure 3 illustrates the use-and-def chain information.



```
S0:      load  r1,5
S1:      add   r1,r2,r3
     ...
S12:     jump  r1
```

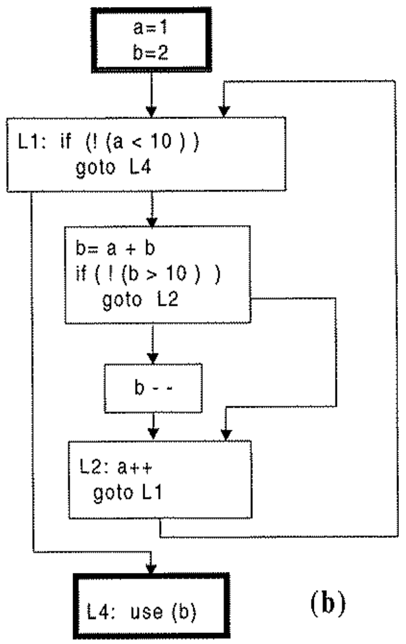
Figure 3. Example Code Segment for Indirect Jumps

Such a use-and-def data analysis is necessary in determining the precise program flow when branch targets are data dependent. It is then clear that, in this case, building the program flow graph is at least as complex as performing the necessary data-flow analysis to resolve indirect branch targets.

It is widely known that many data-flow problems do not have efficient solutions for programs with certain characteristics such as general aliasing [MUCH97]. Some problems have proved to

```
int a, b;
a=1;
b=2;
while(a<10){
   b=a+b;
   if(b>10)
      b--;
   a++;
}
use(b);
```

(a)



(b)

Figure 4. Dismantling of High-level Control Constructs

be NP-complete [LAN921, LAN922, MYER81]. This difficulty in data-flow analysis constitutes the basis of the approach to obfuscation outlined in this section. More precisely, the strategies employed in this work to defeat static analysis are as follows:

- Transform the original program control-flow to that of a data-dependent nature. In other words, the control-flow analysis is transformed into a data-flow problem.
- Increase the complexity of the data-flow analysis to determine the branch targets by incorporating certain program characteristics such as non-trivial aliasing.

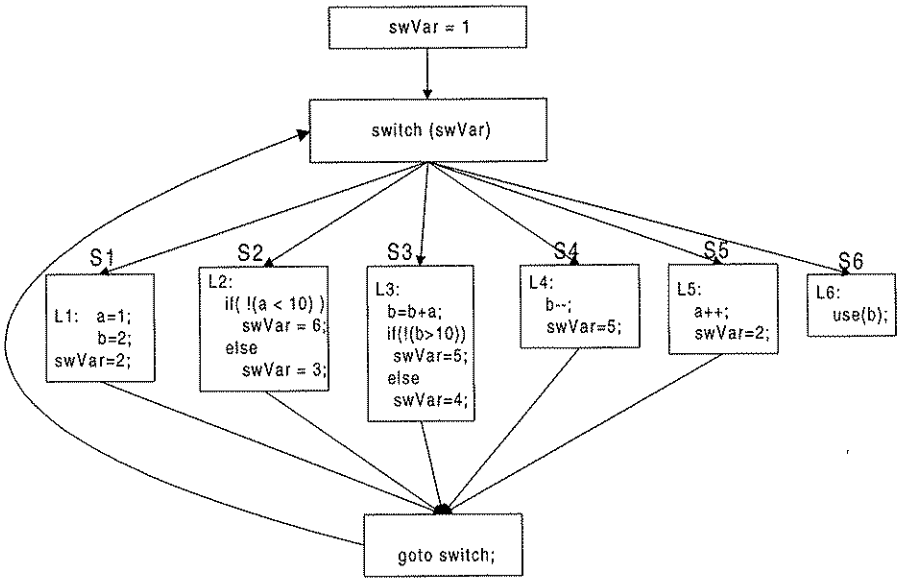
The rest of this section elaborates on the code transformations that implement these strategies.

5.2 Control-Flow Flattening

To make the program control-flow data dependent, we employ a set of code transformations called "control-flow flattening". These transformations are performed in two steps. In the first step, high-level control structures are decomposed into equivalent *if-then-goto* constructs. Figure 4 illustrates such a transform.

In the second step the *goto* statements are modified such that the target addresses of the *goto*'s are determined dynamically. At the source-program level, this transform is modeled by replacing each *goto* statement with an entry to a *switch* statement. The switch control variable is assigned dynamically in each code block to determine which block is to be executed next. An example of such a transform is illustrated in Figure 5.

With these transformations, direct branches are replaced with data-dependent indirect jumps. As a



```
swVar = 1
```

```
switch (swVar)
```

```
L1:  a=1;
     b=2;
swVar=2;
```

```
L2:
  if( !(a < 10) )
     swVar = 6;
  else
     swVar = 3;
```

```
L3:
  b=b+a;
  if(!(b>10))
   swVar=5;
  else
   swVar=4;
```

```
L4:
  b--;
  swVar=5;
```

```
L5:
  a++;
  swVar=2;
```

```
L6:
  use(b);
```

```
goto switch;
```

Figure 5. Transform to Indirect Branches

result, the flowgraph that can be obtained from static branch targets degenerates to a form shown in Figure 6. We will refer to such a degenerate flowgraph informally as *flattened.*

5.3 Introduction of Aliases

After flattening of the program control flow, building the program flowgraph hinges on the complexity of determining branch targets, which is in essence a *use-def* data-flow problem.

In the example shown in Figure 5, the values of the switch control variable `swVar` are assigned dynamically with the constant assignment statements. A constant propagation analysis [MUCH97] combined with use-and-def data flow on the value of `swVar` will quickly reveal, for each block, what the branch target is for the next block, and consequently reveal the entire con-

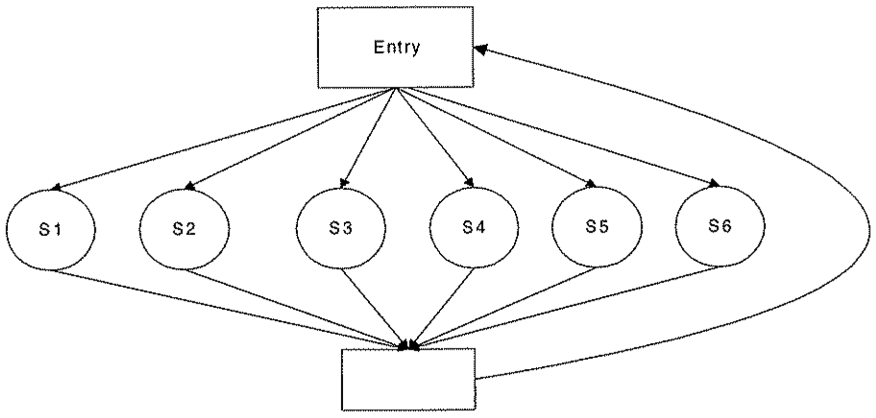


Figure 6. A Flattened Flowgraph

trol-flow graph. To further complicate the static determination of the branch targets, two additional modifications to the program are made: index computation, and introduction of aliases.

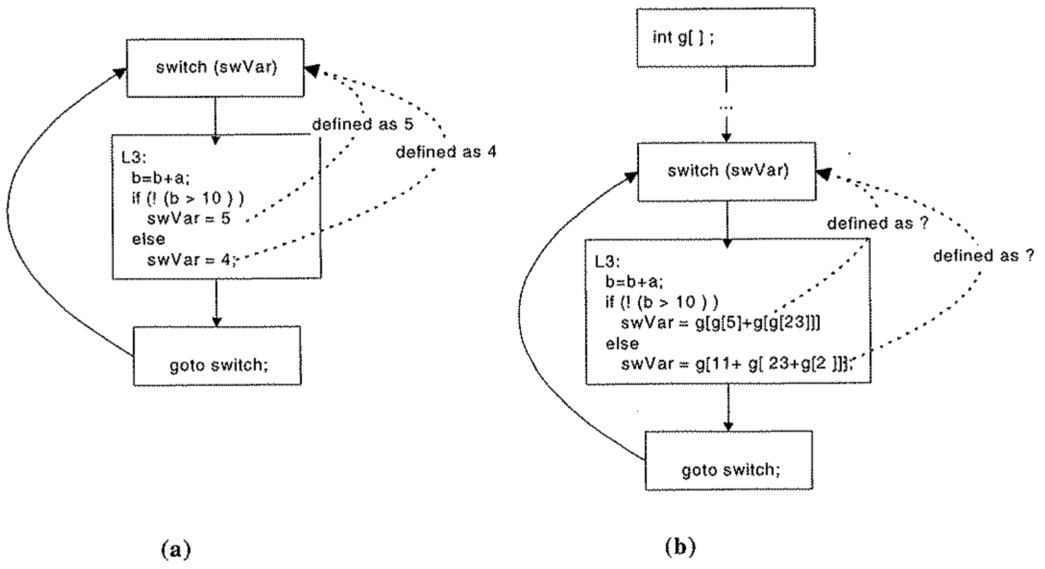
Index Computation: Consider the code segment in Figure 7(a). A use-def analysis on the value of `swVar` (contains branch target information) is straightforward (the dashed line indicates the use-def information chain). Now consider the code segment in Figure 7(b) in which a global array `g[]` is introduced and the value of `swVar` is computed through the elements of the array. Replacing the constant assignments in Figure 7(a) with complex expressions involving array elements implies that the static analyzer must first deduce the array values before the value of `swVar` can be determined.

Some of the array elements contain constant values that are used in the computation of switch variables, while others simply contain arbitrary values. The array elements themselves do not have to remain constant—assignments to the array can be made throughout the program execution as long as it does not interfere with the branch target computation.

For example, consider the scheme in which every *n*th element in the array contains a data value *x* such that:

$$x \equiv 1 \; mod \; j$$

where *j* is a value contained in a specific location in the array (this location can change with every compilation). It is easily seen that any numeric value (such as array subscripts) can be computed using these data values that are *n* elements apart. Throughout the course of execution, the values of the array can be overwritten and *j* can be updated accordingly so that branch target computation can continue undisturbed.



```
switch (swVar)

L3:
  b=b+a;
  if (! (b > 10 ) )
    swVar = 5
  else
    swVar = 4;

goto switch;
```

```
int g[ ];
...
switch (swVar)

L3:
  b=b+a;
  if (! (b > 10 ) )
    swVar = g[g[5]+g[g[23]]]
  else
    swVar = g[11+ g[ 23+g[2 ]]];

goto switch;
```

Figure 7. Example Illustrating Dynamic Computation of Branch Targets

Aggregates, such as arrays or complex data structures, are difficult for static analyzers to process [MUCH97]. Statically analyzing data quantities that have aggregate elements as their values is particularly difficult when elements of the aggregates do not remain static throughout execution. Most static analyzers simply assume that a reference to any element of the aggregate is a reference to the entire structure, and that, potentially, all elements can be changed if an assignment is made to any element. This is the safest and most conservative position.

An algorithm whose purpose is software tampering rather than code improvement would most likely not take this conservative stance. An aggressive data-flow algorithm that is specially tuned to analyze arrays can provide more precise information about arrays and their operations. To deal with the more sophisticated analysis algorithms, the next set of code transformation makes use of a program property that is a primary contributory factor in the complexity of data-flow analysis. This property is aliasing.

Many classical data-flow problems have been proven to be NP complete [LAN922, MYER81]. A fundamental difficulty that data-flow analysis must deal with is the existence of aliases in the program. Precise alias detection in the presence of general pointers and recursive data structures is known to be undecidable [LAN921], and that is the key reason why any data-flow problem influenced by aliasing is fundamentally difficult.

Our second set of transformations focuses on the introduction of non-trivial aliases into the program to influence the computation, and hence the analysis, of the branch targets. These transformations are as follows:

- We first introduce an arbitrary number of pointer variables in each function (this can be parameterized).
- We then insert artificial basic blocks, or code in existing blocks, that assign the pointers to data variables including elements of the global array whose values are used in computing the branch targets.
- Now we can replace references to variables and array elements with indirection through these pointers. Previously meaningful computations on data quantities can be replaced with semantically equivalent computation through the pointers.
- As much as possible, uses of the pointers and their definitions are placed in different blocks. More importantly, assignments to array elements will appear as assignments to the pointer variables which are aliased to array elements.

Some of the basic blocks will execute in all traces of the program, and others are simply dead code. Since the static analyzer does not know which blocks actually execute, and since the definitions of the pointers and their uses are placed in different code blocks, the analyzer will not be able to deduce which definition is in use at each use of the pointer—all pointer assignments will appear live.

Recall the notions of flow-sensitive and flow-insensitive analysis. A flow-insensitive analysis does not depend on the program control-flow information; that is, the analysis perceives the program as a collection of basic blocks, the inter-relations among which are ignored.

It can be shown that the flattened flowgraph in Figure 6 is equivalent to the control-flow perceived by a flow-insensitive analysis [HIND99]. Without knowledge of the branch targets and the execution order of the code blocks, a flow-sensitive analysis cannot provide better precision than a flow-insensitive one.

6 INTER-PROCEDURAL CODE TRANSFORMATION

6.1 Program Call Graph and Inter-procedural Alias Analysis

To compute the effect of aliasing precisely, analysis must be performed at the inter-procedural as well as the intra-procedural level. This is because function invocations can affect aliases in both the calling and the called function. For example, if a global variable is passed as a parameter in a function call, the global variable and the corresponding formal parameter would become aliases inside the called function. Similarly, if an assignment statement inside the called function assigns the address of a global variable to a pass-by-reference parameter, the actual parameter and the global variable will become aliases in the calling function, immediately after the return from the called function.

An inter-procedural data flow analysis relies on the function invocation relationship to determine, among other things, the static information propagation paths in the program, which can then be used to reason about alias relations resulted from function invocations.

The function invocation relations can be encoded in a graph called Program Call Graph (PCG). Formally, a PCG is a triple *(N, e, p)* where; *N* is the set of functions in the program such that $N = \{p_1, p_2, \ldots p_n\}$, p is the function that contains the program entry point; and e is a set of directed edges such that if (p_i, p_j) is an element of e, there exists at least one call from function p_i to p_j.

Figure 8(a) shows an example of a program skeleton in which function f calls g, g calls h and i, and i calls j and g. The PCG for this program is shown in Figure 8(b). The entry function f is marked by the double circle.

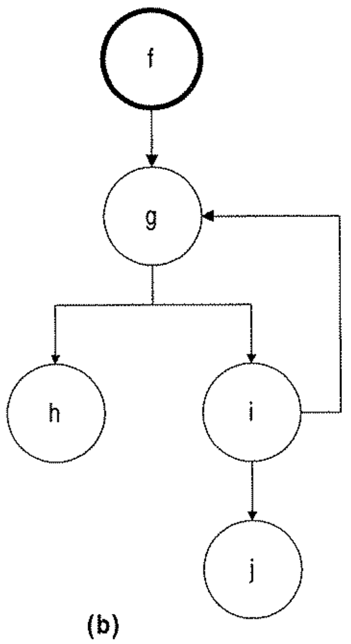
Construction of the PCG is a straightforward process when all function calls are explicit—a simple textual pass over the program will suffice. In the presence of function pointers (or function parameters and function variables), however, a call-site may not be bound to a unique function statically, and therefore the process of constructing the PCG can be more complex.

Several approaches to build the PCG in the presence of function pointers exist, each with varying degrees of complexity and precision [EMAM94]. An approach that simply assumes that an invocation through a function pointer may invoke any function in the program requires a single pass over the program, and thus is the least costly with the least precision. An approach that takes into account only the functions that have been instantiated requires a flow-insensitive traversal of the program, and it generally provides better precision.

A more precise PCG can be obtained by conducting pointer alias analysis prior to building the

```
f( ) {
      ...
      g( );
      ...
 }
g( ){
      ...
      h( );
      ...
      i( );
      ...
 }
i( ){
      ...
      j( );
      ...
      g( );
 }
```

(a)



(b)

Figure 8. Example illustrating Program Call Graph

final PCG. The pointer analysis can help to restrict the number of functions invocable from a call site to the set of functions that are deemed to aliased to the function pointer at that particular point in the program. In other words, construction of a precise PCG hinges on none other than alias analysis of the function pointers.

It should be clear by now that if function pointers are treated in the same manner as other data pointers, the techniques described in section 5.2 can be applied to introduce aliases for function pointers. In so doing we reduce the precision of the PCG representation, and ultimately impede the process of inter-procedural data analysis.

This section describes a set of code transformations based on the above observation. The basic strategy behind these transformations is to modify function calls to include invocations through function pointers. In the mean time, alias-inducing techniques are applied to generate inter-procedural aliases as well as aliases to function pointers. The end result is increased complexity and reduced precision for inter-procedural control flow and subsequent data-flow analyses.

6.2 Function Call Transformations

The presence of function pointer makes the task of constructing precise PCG more difficult. Consider the code segment in Figure 9 in which `func1` calls `func2` and `func3`.

Now consider the code segment in Figure 7, which implements the equivalent functionality.

In Figure 10, `ptr`, `fptr1`, and `fptr2` are function pointers. To discern the target of the indirect call on line S6 in Figure 9, the analysis needs to determine the set of functions to which `fptr1` and `fptr2` are aliased. This requires knowledge of the alias relations that hold on entry to `func1` and possible changes to these relations up to line S6. Without this information, the function pointer `ptr`

```
S1:   func1( ) {
S2:         if ( x > 4 )
S3:             func2( );
S4:         else
S5:                  func3( );
}
```

Figure 9. Example Illustrating Function Calls

on line S6 cannot be bound to any particular function at analysis time.

In this section, we present a set of code transformations designed to modify the program call structure to include invocations via function pointers. More specifically, the transformations include the following steps:

- Unify function signatures to a uniform signature (or a small set of signatures).
- Create function pointers and assign these pointers to the modified functions.
- Create function pointer aliases much in the same way described in the same manner described in section 5.2.
- Modify function calls to invocations through function pointers.

The following discussions consider these steps in turn.

6.2.1 Unify Function Signatures

An important part of the transformations is to create indirect invocations via function pointers. However, if each function in the program has a distinct signature, each invocation must be performed via a pointer of a distinct type[1]. This in itself is not sufficient to fool a static analyzer of any intelligence, for it is a trivial task to match up invocations with different pointer variables. For this reason, prior to creating indirect calls, function signatures must be modified such that all functions in the program conform to only a small number of distinct signatures.

```
S1:   func1(fptr1, fptr2) {
S2:      if (x > 4)
S3:          ptr = fptr1;
S4:      else
S5:          ptr = fptr2;
S6:      ptr;
}
```

Figure 10. Example Illustrating Function Pointers

1. It is possible to use void pointers, but the invocation statement is still distinct, for each function might have different number and types of parameters.

Consider the two function invocations in *func1* in Figure 11. Both function *P* and *Q* return an integer. However, *P* takes one integer parameter while *Q* takes two parameters, one integer and one float. One viable method to unify those two function signatures is to modify *P*'s signature to add a float parameter as depicted in Figure 9. The bold letters indicate the artificial variable and parameter:

```
Func1( ) {              Func1( ) {
   int x, y, z;            int x, y, z;
   float f;                float f1, f2;
   y = P ( x );            y = P (x, f1);
   z = Q ( x, f );         z = Q (x, f2);
}                       }

int P ( int para1) {    int P ( int para1, float para2) {
}                       }

int Q ( int para1, float para2 ) {int Q ( int para1, float para2 )
{
}                       }
      (a)                     (b)
```

Figure 11. Function Signatures Before and After Modification

It should be noted that the number of distinct function signatures in a program is an application specific choice. As an extreme, all functions will have a single uniform signature in which case only one type of function pointer is required. Alternatively, functions can be grouped into a small number of groups, each with a distinct signature. The advantage of the latter scheme is efficiency; In general, a large portion of the functions in a program will contain a short list of parameters with common types (e.g. integers or floats). It is then more economical to group these functions together to form a common signature since the new signature may not be far from any of the original ones, and thus will have less an impact on the run-time space and time complexity.

A number of issues need to be addressed when modifying function signatures and the corresponding invocations. Some of those are discussed below:

- Return type: In order to unify functions with different return types, a *void* type is used in the transformed function signatures. An explicit cast back to the original type at the function invocation completes the transform.
- Complex parameters: Complex parameters such as structures, arrays or functions are replaced with void pointers. Consequently, both function invocations and references to the original parameters inside the function are modified as follows:
  - A direct reference to the original parameter inside the called function is replaced with an indirect reference via the void pointer parameter, following an explicit cast to allow the void pointer to point to the original parameter type.
  - At the function invocation, the original actual parameters are replaced with void pointers

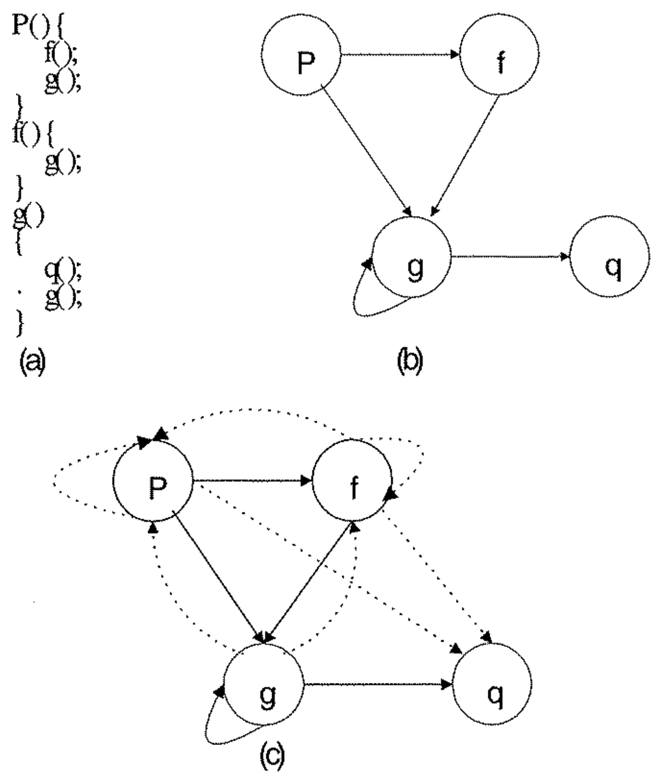
which hold the addresses of the original parameters.

- Parameter list explosion: When unifying functions whose parameter lists do not intersect, it could lead to parameter list explosion. For example, unifying one function with all integer parameters and another one that has all floats will result in twice as many parameters in the unified signature. To avoid parameter list explosion, we impose an upper limit on the number of parameters a function can have. We also employ a naive algorithm to identify functions whose parameter lists heavily intersect with each other—these functions are good candidates for signature unification.
- These transformations result in a large number of functions with an identical signature. The extent to which existing function signatures are transformed is governed by parameters of the transforms (e.g. the number of distinct signatures). Observe that with these transforms, the same function pointer can refer to a large number of otherwise distinct functions.

6.2.2 Create Aliases with Function Pointers and Modify Function Invocations

When function invocations are made through function pointers, the number of potential functions to which a particular call site can be bound statically is equivalent to the set of functions the function pointer is aliased to at the call site.

Using the pointer manipulation techniques described in section 5.2, false alias relations can be introduced to the function pointers. False edges to the PCG, as perceived by a static analyzer, are



```
P(){
   f();
   g();
}
f(){
   g();
}
g()
{
   q();
   g();
}
```

Figure 12. PCG with False Edges

therefore created. In the extreme, all functions can be altered to a single uniform signature. In such a case, the PCG will have a complete degenerate form; that is, any function that contains a function call will have an edge to every other node (function) in the PCG. Figure 9 illustrates such an example. The original call structure is in Figure 9(a) while its corresponding PCG is depicted in Figure 9(b). The post-transform PCG is illustrated in Figure 9(c) where the dashed lines are artificial edges added due to function pointer aliasing.

When a function pointer appears to be aliased to a large number of functions, statically it is impossible to tell exactly which function will be invoked and what data variables are being passed to the called function. This greatly reduces the precision of inter-procedural analysis whose purpose is to discern the information propagation paths among functions.

6.3 Inter-procedural Aliases

Under the conjecture that the PCG of the program is degenerate, and precise information propagation paths amongst functions are not retrievable statically. We describe, in this section, a set of techniques to further thwart static analysis by introducing inter-procedural aliases—aliases whose meaningful resolution requires none other than inter-procedural data-flow analysis.

Inter-procedural aliases are generally created by parameter passing and the accessibility of non-local stack locations. More precisely, the ways in which we introduce inter-procedural aliases are as follows:

6.3.1 Aliases in the called function resulted from the invocation

- Global and local reference aliases: When the address of a global variable or that of its alias is passed to a function, inside the function the global and the corresponding formal parameter become aliases.
- Parameter aliases: Aliases between parameters are introduced in several ways. For example, two pointer parameters become aliases of each other if the same address is bound to both of them, or one is bound to the address of a variable while the other is bound to the address of its alias.

6.3.2 Aliases in the calling function resulted from the invocation

- Alias through return values: If the called function returns the address of a variable visible in the calling function, and the return value is assigned to a pointer variable, the two variables become aliases upon returning from the invoked function.
- Alias through side effects: If the address of a pointer variable is bound to a parameter of a function call, and the parameter is subsequently assigned the address of another pointer variable, also visible in the calling function, the two pointer variables become aliases upon returning from the called function.

Function invocation can result in aliases in both the called and the calling function. This is because inter-procedural information propagation—there is a *forward* parameter binding process in which information propagating from the calling function to the invoked function results in aliasing in the latter. Similarly, there is a *backward* binding process in which information propagating from the called function back to its caller prompt new alias relations in the calling function.

In order to handle the information propagation, both forward and backward, it is crucial that the function invocation information is available in order to bind arguments and parameters properly.

7 Theoretical Evaluation

In this section, we present a set of analytic results to show that the transformations presented in section 4 indeed produce the claimed complexity for program analysis.

7.1 An NP-Complete Argument

We have thus far conjectured that the difficulty of discerning indirect branch target addresses is influenced by aliases in the program. In this section, this claim is supported by a proof which shows that statically determining precise indirect branch addresses is an NP-complete problem in the presence of general pointers.

Theorem 1: In the presence of general pointers, the problem of determining precise indirect branch target addresses is NP-complete.

Proof: The proof consists of a polynomial time reduction from the NP-hard 3-SAT problem to that of determining precise indirect branch targets. This is a variation of the proof originally proposed by Myers in which he proved that various data-flow problems are NP-complete in the presence of aliases [MYER81]. Landi later proposed a similar proof to prove that alias detection is NP-complete in the presence of general pointers [LAN922].

Consider the 3-SAT problem such that

$$\wedge_{i=1}^{n} (V_{i1} \vee V_{i2} \vee V_{i3} , \qquad (1)$$

where $V_{ij} \in \{v_1,...v_m\}$, and $v_1,...v_m$ are propositional variables whose values can be either *true* or *false*. The 3-SAT problem states that it is NP-hard to discern, in a general case, that whether the equation above is satisfiable.

The reduction is shown in the code below. The branch target address is located in the array element A[**true*]. The *if* conditionals are not specified – the assumption is that all paths are potentially executable.

```
L1:   int *true, *false, $**v_1, **v_2, ... **v_m$, *A[];

L2:   A[*true] = &f1();

L3:   if (-)  $v_1 = \&true; \overline{v_1} = \&false$

      else  $v_1 = \&true; \overline{v_1} = \&false$

      if (-)  $v_2 = \&true; \overline{v_2} = \&false$
```

```
        else  $v_2 = \&true; \overline{v_2} = \&false$
        if (-)  $v_n = \&true; \overline{v_n} = \&false$
        else  $v_n = \&true; \overline{v_n} = \&false$
L4:     if (-)  $A[**\overline{v_{11}}] = \&f2()$
        else if (-)  $A[**\overline{v_{12}}] = \&f2()$
        else  $A[**\overline{v_{13}}] = \&f2()$
        if (-)  $A[**\overline{v_{21}}] = \&f2()$
        else if (-)  $A[**\overline{v_{22}}] = \&f2()$
        else  $A[**\overline{v_{23}}] = \&f2()$
        . . .
        if (-)  $A[**\overline{v_{n1}}] = \&f2()$
        else if (-)  $A[**\overline{v_{n2}}] = \&f2()$
        else  $A[**\overline{v_{n3}}] = \&f2()$
L5:
```

The code segment L1 declares the variables and an array A[]. $v_1, v_2, ...v_m$ are doubly dereferenced pointer variables. L2 assigns *A[*true]* to the address of *f1*.

A path from L3 to L4 represents a truth assignment to the propositional variables for the 3-SAT problem. In this code, the assignment to *true* is represented as an alias relationship $<*v_i, false>$, and the alias $<*v_i, false>$ represents assigning *false* to variable v_i.

If the truth assignment for the particular path from L3 to L4 satisfies the 3-SAT formula, then every clause contains at least one literal that is true. This means that there exists at least one path between L4 and L5 on which the value of A[*true] is never reassigned. Consider choosing the path that goes through the true literal in every clause, and in every clause it assigns *A[*false]* to f2() since every variable $*v_{ij}$ on that path is aliased to *false*.

If the truth assignment renders the formula not satisfiable, then there exists at least one clause, ($V_{i1} \vee V_{i2} \vee V_{i3}$), for which every literal is *false* (i.e., all the literals in the clause are aliased to *false*). This implies that $*\overline{v_{ij}}$ is aliased to *true* for this clause. Because every path from L3 to L4 must go through the following statement:

```
If (-) $A[**\overline{v_{i1}}] = \&f2()$
        else if (-) $A[**\overline{v_{i2}}] = \&f2()$
    else $A[**\overline{v_{i3}}] = \&f2()$
```

Therefore, at program point L5, $A[*true]$ must point to the address of *f2*.

The above code segment shows that 3-SAT is satisfiable if and only if the branch target address contained in $A[*true]$ is the address of *f1*. This proves that 3-SAT is polynomial reducible to the problem of finding precise indirect branch target addresses.

7.2 The Parameters that Affect Alias Analysis

The NP-completeness proof presents a complexity measure for analysis of general case post-transform programs. However, such a proof does not guarantee the average case complexity—for any given program, the complexity of analysis could be well within the grasp of a static analyzer (because of its size, characteristics, etc.)

In this section, we examine the complexity measures for practical alias analysis. In practice, alias analysis is conducted in an approximation manner—the results reported by the alias analyzer may not be precisely the same as the actual alias relations, but usually contain a conservative estimate of the reality (a super set of the actual alias relations). How far the reported results are from the actual alias set is called the *precision* of alias analysis.

The discussion here is concerned with the efficiency of the alias analysis as well as its precision. An analysis algorithm may be able to reach a conclusion quickly if precision is not of major concern. A clear example is to trivially report that every variable is aliased to every other variable—an analysis would take very little time but would report with the least precision.

The complexity of alias analysis has been examined at a great length in various studies [HIND99, LAN922, LARY92]. In the discussion here, we distill from the previous research the essential parameters that affect alias analysis and explore the effect of the code transformations with respect to these parameters.

Before delving into the details of analysis, we introduce a number of conventions and terminology that are used in the subsequent discussions.

"Points-to" representation: The discussion hereafter is based on the *points-to* abstraction of alias information [EMAM94]. Using this representation, *x* points-to *y* if *x* contains the address of *y*. The "Points-to" representation is commonly regarded as more compact and efficient than the explicit alias pair representation [HIND99]. For example, the alias relations shown in Figure 10 can be represented with two points-to tuples <*a, b> and <*b, c>, while in the more traditional explicit representation, this alias graph is represented by <*a, b>, <**a, c>, <*b, c>, and <**a, *b> [1].

1. The relationship <**a, c> and <**a, *b> can be inferred from <*a, b> and <*b, c> of the points-to representation

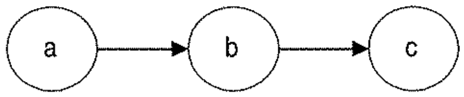


Figure 13. Example Alias Graph

Complexity Variables: Listed below are the variables that used throughout the complexity analysis discussion.

1. *NL(p)* is the set of non-local variables that are visible in function p. In a C-like language, *NL(p)* is the set of global variables.
2. *LOCAL(p)* is the set of local variables of function p.
3. $LOCAL_{av}$ is the number of average local variables of all functions.
4. *PARAM(p)* is the set of formal parameters of function p.
5. *PARAM(p, i)* is the ith formal parameter of function p.
6. *ARGU(q, p)* is the set of arguments of call site q that calls p.
7. *ARGU(q, i, p)* is the ith argument of function call site q that calls p.
8. *PARGU(q, p)* is the set of pointer arguments of call site q that calls p.
9. $PARGU_{av}$ is the average number of pointer arguments of all function calls.
10. *ALIASE(a, d)* is the set of aliases of object a after d level of dereference.
11. *AR* is the number of alias relations currently holding.
12. AR_{max} is the maximum number of alias relations holding at any time.
13. AR_{av} is the average number of alias relations holding throughout execution of the program.
14. *F* is the number of functions in the program.
15. *S* is the number of pointer assignment statements in a function.
16. *FC* is the number of function calls in the program. In the presence of function pointers, *FC* can be larger than the number of physical call sites in the program.

7.3 Intra-procedural Analysis

Determining the range of possible aliases in a program is essential to decipher the behavior of the program statically—where and how a variable might be accessed carries information about the algorithm the program employs, and therefore is important to intelligent tampering or impersonation attacks.

This section examines the complexity of intra-procedural alias analysis in the presence of the flatten-n-jump technique described in section 4. Intra-procedural analysis requires the examination of the statements within a function[1] and the subsequent combinatorial analysis (if any) over their effect on aliasing. To be more specific, the intra-procedural phase entails the following operations:

- The processing of each pointer assignment statement—the effect of this particular statement has on the alias relations.
- Analysis of the combinatorial effects of all the statements (either in a flow-sensitive or insensitive manner)

The intra-procedural phase of the analysis may be repeated multiple times, for the inter-procedural analysis may result in new information that need to be processed before the alias set converges. The discussion in this section is initially focused on intra-procedural analysis only, assuming the information propagated from other functions remain static.

7.3.1 Processing of pointer assignment statements

The only kind of statements where alias information might be modified is pointer assignments of the form:

$$Ptr = Expression(Q)$$

where the evaluation of *Expression(Q)* returns an address of a memory location, which is subsequently placed in the pointer object *Ptr.* Assuming *p* and *p'* are the program points immediately before and after the pointer assignment statement, and *AR* the set of alias relations holding at *p*, processing the assignment statement is to produce the set of alias relations *AR'* holding at *p'*, given the effect of executing the statement (see Figure 14).

There exists a set of well-known transfer functions f that handles different statements, some are more complex than others [MUCH97]. These transfer functions specify a set of rules via which alias relations are modified. For example, the transfer function for the statement p = q where p and q are both pointers is such that,

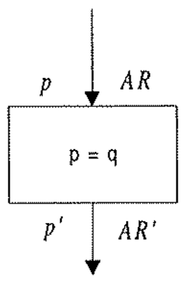


Figure 14. Processing of a pointer assignment statement "*p* = *q*"

1. This will be done in the order of execution if flow-sensitive algorithms are used.

$$AR' = AR - (any\ alias\ of\ {*}p) + ({*}p,\ deref(q,\ 1))$$

This transfer function kills any alias of *p (since p is reassigned) and adds the alias of *p and anything q points to (since p is now pointing to whatever q points to). In Figure 14, if *AR* = {<*p, a> <*p, b> <*q, c>}, after the statement *p = q*, *AR'* is {<*p, c> <*q, c>}, the alias relations <*p, a> <*p, b> are killed by the statement.

More generally, for a statement of the form

$$Pi = Qj \qquad (1)$$

where *Pi* denotes a pointer object *i* levels of dereference away from *P*, and *Qj* is a pointer object *j* levels of dereference away from object *Q*, the transfer function is described in Figure 15.

```
Transfer Function f {
if (deref(p,i) must alias to pointer object c)
   AR' = AR - (any alias relation of c) + (*c, deref(q, j+1))
else
   AR' = AR + { <*a, b> | a is in deref(p, I)
                              and b is in deref(q, j+1) }
```

Figure 15. Transfer Function for Statement 1

The `deref(p, i)` function returns the set of objects that are *i* levels away from object *P*. For example, for the alias graph in Figure 13, `deref(a, 2)` will return object `c`. The algorithm of `deref(p, i)` is based on the points-to representation, and it is described in Figure 16.

```
AliasSolution <-- { }
function deref (p, derefLevel) {
   if (derefLevel = 0)
      AliasSolution <-- AliasSolution + p;
   else
      for each alias relation (*p, target) or (p, target), do
         deref (target, derefLevel -1)
      end do
   end if }
```

Figure 16. Algorithm for dereferencing pointer variables

This algorithm is similar to the alias query algorithm presented in Hind et al. [HIND99]. The average and worst case time complexity of such an algorithm is as follows, respectively,

$O\ (ARsg_{av}{}^{derefLevel})$, and

$$O(ARsg_{max}^{derefLevel})$$

where $ARsg_{av}$ and $ARsg_{max}$ denote the average and the maximum number of alias relations for an object due to a single level of dereference. Most pointer assignment statements can be decomposed into statements of the form shown in (1), for which two calls to deref are made. The result from calling deref is an alias set whose size is bound by *ARmul,* and paring the results of calling two deref is

$$O(ARmul * ARmul).$$

Therefore the complexity of applying the transfer function to process a pointer assignment statement is roughly,

$$O(ARsg_{max}^{derefLevel} + ARmul_{max} * ARmul_{max}),$$

for the worst case, and

$$O(ARsg_{av}^{derefLevel} + ARmul_{av} * ARmul_{av}),$$

for the average case.

7.3.2 *Combinatorial Analysis*

In a flow-insensitive algorithm, the effect of individual statements on aliasing is simply summarized together to discern the effect of a function on aliasing. Using such an algorithm, the worst case complexity of one intra-procedural analysis phase is,

$$O(S_{max} * F * ARsg_{max}^{derefLevel} \quad ARmul_{max} * ARmul_{max}),$$

where S_{max} denotes the maximum number of pointer assignment statements in a function, and F denotes the number of functions in the program. In a flow-sensitive analysis, the program control-flow is taken into account in the analysis. One of the most significant differences between a flow-sensitive analysis versus an insensitive one is the treatment of the combinatorial effect of individual statements on aliasing. The flow-insensitive analysis summarizes over all statements, regardless of the order of execution. A flow-sensitive analysis, however, combines alias relations at such program points called *meet nodes* [HIND98, CHOI91]. Meet nodes are actual or abstract nodes in the flow graph where different flows meet. For example, in Figure 17, node c is a meet node where the yes and the no branch of the if statement meet.

At each meet node, a union operation on the alias relations is performed, and the complexity of that operation is bound by, again, the maximum number of alias relations holding at any time during execution. This union operation is

$$O(\text{number of meet nodes} * ARmax)$$

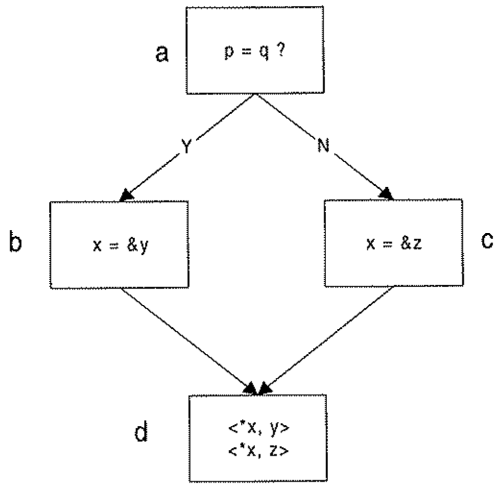


Figure 17. Example of a Meet Node

If the flow graph is fully flattened, there exists only one meet node in each function—the switch statement node (see Figure 5). The complexity of this analysis step is therefore straightforward:

$$O(F * ARmax),$$

where *F* is the number of functions and *ARmax* denotes the largest number of alias relations holding at any execution point. Thus, the complexity of one flow-sensitive intra-procedural analysis phase can be described as,

$$O (S_{max} * F * AR*g_{max} + F^{derefLevel} * ARmax),\ ARmul_{max}\quad ARmul_{max}$$

7.4 Inter-procedural Alias Analysis

As described in section 4.4, inter-procedural alias analysis primarily handles information propagation between the calling and the called function. In particular, two information propagation processes are needed in order to discern the effect of function calls. They are: *forward binding* and *backward binding*.

Forward binding maps the set of aliases holding at a call site in function *f* into aliases holding on entry into the called function *g*, using the argument/parameter mappings of the function invocation. Backward binding maps aliases holding at the exit of the invoked function g into aliases holding immediately following the execution of *g* in *f*.

This section investigates the complexity of the forward and backward binding process, for they directly affects the efficacy of the inter-procedural code transforms. The complexity discussion continues in the context of a C-like language with pass-by-value parameters and pointers.

For every function call, the forward binding is shown in Figure 18. This forward binding algo-

```
ForwardBinding( ) {
   for each pointer variable a in ARGU(q, p) such that
                                            a is ARGU(q, i, p), do
      replace <*a, x> (or <a, x>) in AR with <*PARAM(p, i), x>
                                            (or <PARAM(p, i), x>)
   end do
end forwardBinding}
```

Figure 18. Forward Binding Algorithm

rithm simply replaces all occurrences of an actual pointer parameter in the alias relation with the corresponding formal parameter. For example, performing the forward binding process for the function call in Figure 19 would result in {<*m, a>, < *n, a>} holding at the entry of function g.

The forwarding binding algorithm in Figure 18 is devised based on the points-to alias abstraction, and it is otherwise general in the sense that the complexity of the algorithm does not depend on any implementation of data structures specific to the algorithm. The average and worst case time complexity for the forward-binding process are such as follows:

$$O(PAGU(q, p) * AR_{av}), \qquad \text{and}$$

$$O(PARGU(q, p), AR_{max})$$

where *PARGU(q,p)* is the set of pointer arguments at call site *q* that calls *p*, and ARav, ARmax denote the average and the maximum number of alias relations holding during execution.

The backward binding process, in the presence of pass-by-value and pointer arguments, is described in Figure 20.

The backward binding process simply discards each alias relation that involves a local variable in the called function, and replaces alias relations of the form <*PARAM(p,j), x> with <*ARGU(q, i

```
int a,b;
   f() {
      int *x, *y;
      x  =  &a;
      y  =  &a;
                        ------<*x, a> <*y, a> hold at this point
      g(x, y);
   }

   g(int *m, int *n) {
      int *x;
      x  =  &b;
      n  =  x;
      *m =  *x + 1;
   }
```

Figure 19. Example Illustrating Forward Binding

```
BackwardBinding ( ) {
   for each(x, y) in current alias relation AR, do
      if either x or y is in LOCAL(p),
         discard (x, y) from AR
      else
         replace (x, y) where x is PARAM(p, i) with (ARGU(q, i, p), y)
      end if
   end do
}
```

Figure 20. Backward Binding Algorithm

p), x>. Time complexity of the backward binding process is, O (ARav * LOCAL(p)), or O (ARmax * LOCAL(p)) for worst case Consider again the example in Figure 5.7. The set of alias relations holding at the exit of function g is {<*m, a>, <*n, b>, <*x, b>). Performing the backward-binding algorithm would result in elimination of the alias relation <*x, b> since x is only local to g. The alias relations holding at the return from g are, { <*x, a>, <*y, b> } For each function call, the cost of propagating alias information back and forth between the called and calling function is,

$$O\,(\,ARav * (LOCAL(p) + PARGU(q, p))\,, \qquad \text{and}$$

$$O\,(\,ARmax * (LOCAL(p) + PARGU(q, p))\,, \qquad \text{for the worst case.}$$

Thus the overall time complexity in handling inter-procedural information propagation is:

$$O\,(\,FC * ARav * \;(LOCALav + PARGUav)\,) \qquad (2)$$

where FC is the number of edges in the PCG.

The inter-procedural binding process is further complicated by the fact that function calls can happen recursively. Whenever recursive function calls are involved, the backward binding process needs to be more conservative in order not to result in incorrect analysis outcome. For example, when a calling function is also visible from a called function due to recursion, the alias relations concerning only local variables of the called function may carry into the calling function. Therefore, the backward binding process must distinguish which local variables might be visible to the calling function (perhaps passed with an "&" operator), and care must be taken that these alias relations do not get discarded upon exit from the called function.

7.5 Iterations Over the PCG

Described in section 7.3 and 7.4 are complexity measures for intra and inter-procedural alias analysis. The complexity formulas (1) and (2) represent one-time cost for conducting intra and inter-procedural analysis. However, an aggressive analysis algorithm might attempt to iterate over the PCG and repeat the process of (1) and (2) multiple times before the alias set converges. Using such an iterative algorithm, the maximum number of iterations over the PCG is

$$O(ARmax * F),$$

where *ARmax* is the maximum number of alias relations holding at any time and *F* is the number of functions. During each iteration, the intra-procedural and the inter-procedural phase might be repeated; that is, the overall complexity of iterating over PCG and repeating the intra and inter-procedural analysis is,

$$\begin{aligned} &O((ARmax * F) * ((FC * ARmax * (LOCALmax + PARGUmax)) \\ &\qquad + (S_{\max} * F * ARsg_{\max}{}^{derefLevel} + ARmul_{\max} * ARmul_{\max} + F * ARmax)) \end{aligned} \qquad (3)$$

7.6 Putting Together the Complexity Argument

The complexity measures represent the worst-case time requirement for a full-up alias analysis. As shown in (3), what is of consequence is the size of the alias set *AR*, the size of the parameter set (*PARGU* and *LOCAL*), and the size of the program (*S* and *F*).

Worst-case complexity, however, is often not an accurate indicator of performance. In practice, what is of interest is the number of times a function is visited and the number of times a transfer function is evaluated during analysis. For a program with a flattened control flow, a degenerate call structure and an abundant number of aliases, a function can be visited each time a function call site is encountered (consider a fully connected program call graph), and a basic block will be analyzed each time a branch instruction is met (consider a fully connected flow graph). In such a case, the number of times a particular function is visited and a pointer assignment statement is analyzed is not far from the worst case estimate. Such an analysis will eventually converge and report a set of results that include as many spurious alias relations as it possibly can: in several occasions when tested against automated analysis algorithms, the algorithm halts with a report that every pointer variable is aliased to every memory location that appears on the right hand side of some assignment statement.

We have finally reached the major result of the transformations that we have performed on programs:

> *The degeneration of the program control flow and call structure renders a data-dependent program flow structure. That is, control-flow and data-flow analysis are made co-dependent.*

The result of this co-dependence are: (1) vastly increased complexity for both control-flow and data-flow analyses, and (2) reduced analysis precision. The practical result is that automatic static analysis of a transofrmed program is essentially impossible.

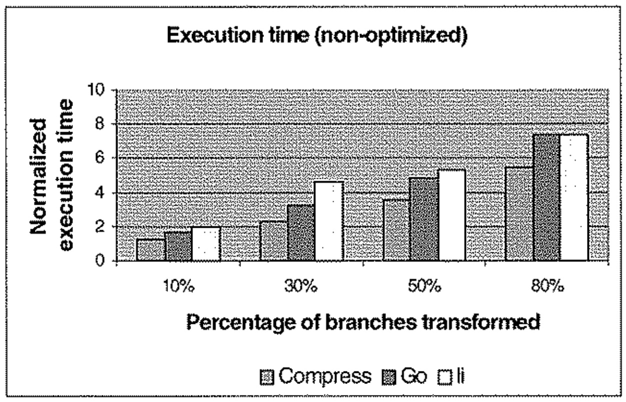
8 Performance Results and Empirical Evaluation

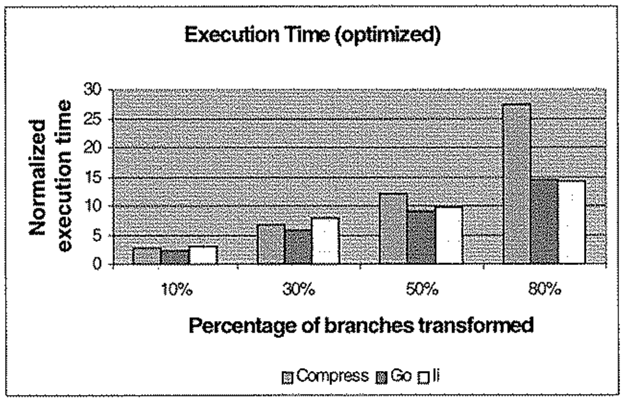
In this section we report performance results obtained with experimental transformations on the SPEC95 benchmark programs. Of issue here are three measures: *performance of the transformed program*, *performance of static analysis*, and *precision of static analysis*.

By performance of the transformed programs, we mean the execution time and the executable object size after transformation. These measures reflect the cost of the transformation. By performance of static analysis, we mean the time taken for the analysis tool to reach closure and terminate. A related but equally important criterion is the precision of static analysis, which indicates how accurate the analysis result is compared to the true alias relationships.

8.1 Performance of the Transformed Program

The following data was obtained by applying our transforms to SPEC95 benchmark programs. Three SPEC programs were used in this experiment, *Compress95*, *Go* and *LI*. Go is a branch-intensive implementation of the Chinese board game GO. Compress95 implements a tightly-looping compress algorithm, and LI is a LISP interpreter program. These programs embody a wide range of high-level language constructs, and therefore are good representatives of real-world programs.





(a) (b)

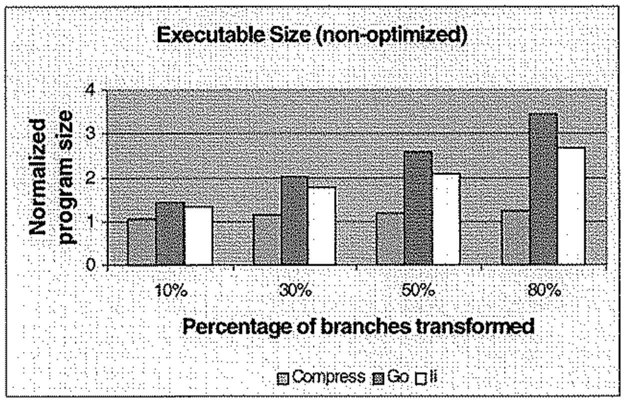
Figure 21. Execution time of the transformed benchmark programs

We conducted experiments on both optimized (with the gcc -O option) and non-optimized versions of the programs. The experiments were executed on a SPARC server. Results from the experiments are depicted in Figure 21.

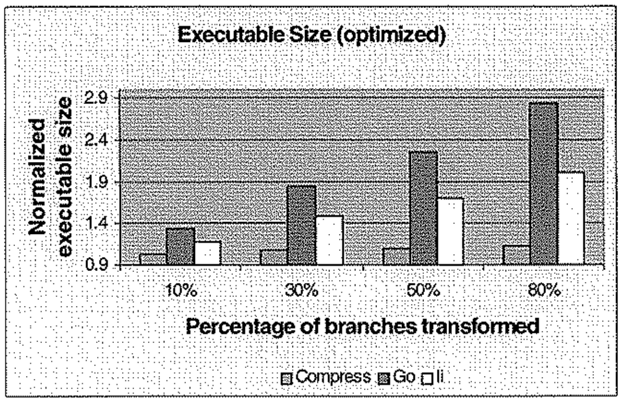
As the performance data show, in both optimized and non-optimized cases, the performance-slowdown increases exponentially with the percentage of transformed branches in the program. On average, performance-slowdown is significantly worse in the optimized case. This is encouraging because what we observed was that our transformation considerably hindered the optimization that the compiler is able to perform.

The performance of Go and LI were similar for both optimized and non-optimized code. Of all three untransformed programs, compiler optimization performed best on Compress95—a whopping 80% decrease in the execution time due to optimization. However, as can be seen in Figure 21(a), our transforms removed significant optimization potential from Compress95; the execution speed of the transformed and optimized Compress95 diverges most significantly from that of the original optimized program. As Compress95 is a loop-intensive program, it is likely that certain analysis which enabled significant loop or loop kernel optimization was no longer possible after our transform was performed.

The executable object size of the three benchmarks grew with increased branch replacement. The experimental data of program size expansion is shown in Figure 22. Go, a branch-intensive program, displays the largest code growth with our transform. For 80% replacement of direct branches, the executable size increased by a factor of 3 for Go and LI, and by roughly 10% for Compress95. Compress95 contains relatively fewer static branches which resulted in less code growth with the transforms.



(a)



(b)

Figure 22. Program size of the transformed benchmark programs

We believe that these results are representative of many programs. It appears that replacing 50% of the branches will result in an increase of a factor of 4 in the execution speed of the program. At the same time, the program will almost double in size. In these experiments, we used a random algorithm to choose which branch to transform. An obvious improvement is to employ intelligence to do the following: a) identify the regions of the program that require greater protection from static analysis, and b) selectively perform transformation on the less-often-executed branches for better performance penalty. Trade-offs between these two criteria need to be considered for the most effective solution.

8.2 Performance and Precision of Static Analysis

In this experiment, we report the result of running existing static analysis tools. The notable static analysis tools include the NPIC tool from IBM [HIND99] and the PAF toolkit from Rutgers university [PAF]. The NPIC tool implements a sophisticated algorithm, and represents the state-of-the-art in the field of static analysis. Unfortunately, IBM no longer maintains and distributes the tool. Our empirical results therefore are obtained using the PAF toolkit. PAF implements a flow-sensitive, inter-procedural pointer analysis algorithm [LARY92]. Both NPIC and PAF perform control-flow analysis exactly once with no further refinement on the flow graph.

In our experiments, *before* transformation PAF successfully analyzed small, toy programs but failed to handle some of the large programs included in the SPEC benchmarks. We tested PAF on a wide range of small programs that contain extensive looping constructs and branching statements. In all of the test cases, PAF terminated reporting the largest possible number of aliases in the program (the worst possible precision)—in a program with n distinct pointer assignments and k basic blocks, it reports $n*k$ alias relations. Because of the size of the test programs, we observed negligible differences in the pre- and post-transformation analysis time. The experience with the PAF tool, albeit with limited test cases, indicated that PAF failed to resolve aliases across the flattened basic blocks, and that our technique of making data-flow and control-flow co-dependent presents a fundamental difficulty that existing analysis algorithms lack the sophistication to handle.

9 IMPLEMENTATION

In this section we briefly discuess the implementation of a prototype of the One-way Translation mechanism. We identified three design goals for the One-way Translation compiler. They are:

- Platform-independence
- Modular support for code transforms
- Easy extension to other high-level languages

To meet these design goals, we selected ANSI C as the target language and the SUIF compiler from Stanford as the infrastructure to implement code transformations[SUIF].

The SUIF compiler toolkit offers an extensive set of utilities for source-code manipulation. The current SUIF system uses an intermediate representation called SUIF and a set of well-defined functionality to manipulate the SUIF representation. It takes as input a C program, performs code transformations, and produces as output a new ANSI C program. The resulting C code can then be compiled using any ANSI C compiler. Since the code transforms operate on the SUIF representation which is largely architecture independent, these choices meet the first requirement, platform-independence.

The types of code transformation may evolve as new protection techniques surface. It is therefore important to have modular support for incorporating additional transformations. In SUIF, code transformations are implemented as SUIF passes, and SUIF provides a programming environment that allows the independent development and easy integration and inter-operation of different passes. This made SUIF an ideal implementation environment for the One-way Translation compiler.

Finally, source-to-source translation tools exist between ANSI C and several other languages such as C++, Pascal and Fortran. Programs written in those languages can be translated into ANSI C before undergoing the code transforms. Therefore the choice of ANSI C as the target language allows easy extension to other languages. The SUIF team is currently designing a JAVA front-end, which when available, can be incorporated into out One-way Translation compiler to handle JAVA programs.

At this level of automation, the programmer is required to specify transformation parameters such as the level of degeneration and aliasing. Fortunately, this step is not terribly burdensome since it involves no more than setting the values of a few parameters.

10 Correctness Discussion

When programs are transformed during compilation, there is always the issue of whether the transformation is performed correctly—whether the resulting code preserves the semantics of the original program.

Proving the correctness of the transforms formally is beyond the immediate scope of this work. In fact, the compiler community has not solved the general correctness question regarding the more traditional code transformations performed by optimizing compilers. While the correctness issue is not dealt with directly in this work, we point to a few research ideas that address the various aspects of the correctness problem.

Translation validation [PSS98] is a technique designed to check the result of each compilation against the source program and pinpoint errors on-the-fly. Necula proposed a practical framework for translation validation within which small instances of code transforms (described as a pass) can be validated [NECU00]. Necula showed that one can implement a practical translation validation infrastructure with about the effort typically required to implement one compiler pass. Since our transformations are implemented as compiler passes, we believe that while we may not be able to prove that our compiler is always correct, we can at least check the correctness of each compilation using translation validation techniques.

11 Defending against Dynamic Analysis

Dynamic analysis of programs include execution simulation, profiling and debugging. At the writing of this document, the mechanisms to defend against dynamic analysis are very much work-in-progress, and they are documented below.

Our One-way Translation technique applies randomizations to the program that are designed specifically to induce analysis complexity. The randomness introduced in the program can defend against dynamic as well as static analyses. The following paragraphs explain why this is the case.

Dynamic analysis of an executing program can be viewed as a special case of software testing. The general problem of comprehensive program testing is known to be difficult [ABK88], for the number of potential paths through the program execution can be exponential in the size of the program. Comprehensive testing, therefore, faces the issue of completeness—whether the test cases cover all possible paths.

If we take this observation a step further and consider it in the context of software protection, dynamic analysis of programs can be effectively thwarted if the following are true:

- Dynamic analysis of program for security purposes is as difficult as conducting a comprehensive path testing for the program.
- The number of potential paths through the program is made intentionally large such that testing the program will be essentially difficult.

Another potentially fruitful idea is to explore non-determinism in programs. Multi-threaded programs, for example, exhibit non-deterministic behavior, which is problematic for program simulation. If the concurrent threads execute in a random order, and there are a large number of possible execution scenarios due to concurrency, the chance of simulating the program and actually learning something definitive and useful about the current execution is fairly small.

Non-determinism is a powerful and promising technique which can change the control flow of the program in a substantial way—the reverse mapping from the concurrent binary threads back to the sequential control flow of the source code will be extremely difficult. However, it is not a trivial task to convert sequential programs that are not originally designed for concurrency into concurrent programs. We will investigate the feasibility of such an approach in pursuing two possible directions—restructuring the program into a set of cooperating tasks and adding benign tasks that provide minimal yet useful service. We suspect that some hybrid of the two will have to be adopted.

All these ideas are rooted in the same principle of state inflation. That is, the internal state space of the program is made intentionally large such that any realistic analysis cannot be completed within reasonable amount of time. This can be explained using a simple state machine model of programs.

If we view programs as state machines, state transitions are prompted by program execution. Each instruction executed emits some information into its environment that can be gathered and used in program analysis. Most real world programs have a limited number of states, and program execution typically revisits these states (e.g. in a loop). If an adversary is able to discern that a previously visited state is entered again, they will know how execution will proceed after that (simply repeating the set of states since the last visit). Inflating the state space does not imply a blind addition of states. However, it involves a careful reorganization of the state space such that the first occurrence of a recognizable state does not occur within a prescribed time frame. A potential complication is the performance issue once the state pace is increased. Care should be taken so that it does not seriously hinder the performance of the program.

Dynamic profiling and debugging is yet another technique for program analysis. For example, once a control-flow graph (CFG) is built, a clever trick is to group the basic blocks and edges of the CFG into equivalence classes based on frequency of execution determined by the profiling analysis. It is then trivial to map one CFG onto another, assuming the two graphs do correspond to each other.

This type of analysis can be thwarted by adopting techniques that confuse the profiling algorithm deliberately. For example, benign code that does non-critical but useful computations can be added to the program to muddle with the execution count obtained by sampling of the program counters. In addition, spurious computations on consequential variables can be used to confuse the profiling process.

12 RELATED WORK

Protecting trusted software from untrustworthy hosts has many potential applications. The copyright protection industry has long employed some of the methods presented in this document. Another context in which this problem is of interest is the mobile-code environment [FGS96, MEAD97, YEE97]. Mobile programs traverse from machine to machine, and in some cases it is necessary to ensure that the integrity of the program is preserved by the chain of executing hosts.

A few studies have been reported that were concerned with the protection of mobile agents from malicious hosts. Most of the work is done to protect Java byte code [CTL97a, CTL97b], and these studies have typically concentrated on code-obfuscation techniques. While these techniques are innovative in their own right, they lack the basis of formal analysis and provable results. Most of these techniques are based on loosely-defined software complexity metrics that bear no real correspondence to program understandability. For example, one of the complexity metrics used in evaluating how well the code has been obfuscated considers that two-dimensional arrays are more complex than arrays of only one dimension [MUNS93]. However, this overlooks the fact that in some cases, it is more intuitive and simpler to represent the data in a multi-dimensional array than otherwise. These techniques also assume that the attacker does not have access to the source code while we assume a very sophisticated adversary who knows all there is to know about the source code before the transformations. These obfuscation operations occur only at the source code level while we employ randomization and obfuscation techniques at all stages of the code generation – from source code to run-time.

One particularly relevant area is that of mobile cryptography [SATS98]. This work raises the possibility that programs could be executed in an encrypted form. The approach is able to provide code secrecy and integrity automatically, but the technique, as it stands now, only applies to polynomial and rational functions. It remains to be seen if it can be extended to work with general-purpose programs.

Our problem context is fundamentally different from that of a mobile code environment. First of all, if a mobile program fails to survive, it will not in general lead to system-wide damage. Second, mobile code may execute on any arbitrary platform that it might or might not know anything about. In that sense, building survivable mobile code is more difficult than the problem we set out to solve. In our problem context, we have some control over the host machines, although we have to allow the possibility that these machines can be compromised. This implies that we can make more precise assumptions about the executing environment. Our one-way translation techniques can be extended to the application software executing on the host machine if necessary, while mobile agents will not be able to change anything on the host machine. We believe that viable techniques can be devised to build secure survivability schemes, while solving the corresponding problem in the mobile environment may be an unattainable goal.

Aucsmith's work on the Integrity Verification Kernel (IVK) [AUCS96] at Intel is of direct interest because it puts forth the concept of building tamper-resistant software. An IVK consists of multiple cells (code segments), and all the cells are encrypted except the one that is currently executing. The executed cell becomes encrypted again after execution while the next cell decrypts. This method requires significant computational resources to accomplish the dynamic encryption and decryption process. In addition, cumbersome manual intervention on the part of the programmer is needed in order to identify critical code segments that must be specially armored to create IVKs in the first place.

The Immunix project at Oregon Graduate Institute includes an effort to build survivable operating systems through diversity specialization [PBCW96]. The concern in that work is mainly to thwart class attacks due to software flaws. The claim is that the specialization method allows the system to guard the validity of the operating-system software, both statically and dynamically. It is unclear, and the designers of Immunix provided little hint, whether diversity techniques can be as effective for complex software such as operating systems. Immunix's stack guard work is also of interest to us, but of a slightly different kind. Instead of placing canary words in the stack frame to detect buffer overrun attacks, our method would call for a randomization in the placement of return addresses to prevent buffer overrun attacks from happening in the first place.

Another approach worth noting is Devanbu and Stubblebine's work on protecting stack and queue integrity on hostile platforms [DEST98]. Their method uses digital signature chains, starting from an initial signature that is protected on a trusted device, to verify the integrity of the data and data operations. A trusted device handles the signature computation and data generation, while the hostile host manages the data structure. This work does not address methods for protecting general-purpose software.

13 SUMMARY

Motivation of this work initially stems from the important issue of protecting network survivability mechanisms. The proper protection of such mechanisms is essential but difficult because parts of the mechanism may operate on the untrustworthy application hosts.

Our solution to this problem is to inhibit program analysis by a suite of mechanisms including One-way Translation and diversity schemes. The translation is made one way by embedding randomness in the translation process. Our approach precludes program analysis under many circumstances, or at least makes it an extremely expensive operation.

One-way translation is not a perfect security mechanism. There are scenarios in which an adversary could defeat the protection provided but we claim that they are the most unlikely to occur because they require extreme system access. If such access were possible (because of insider access) then there are many simpler ways for the adversary to proceed. One-way translation is a compromise based on maximizing the protection provided for what is expected to be reasonable cost.

14 REFERENCES

[AETA97] Anderson *et al. Continuous Profiling: Where Have All the Cycles Gone?* ACM Transactions on Computer Systems, November 1997, Vol 15, No.4.

[ABK88] P. Amman, S.S. Brilliant, and J.C. Knight, *The Effect Of Imperfect Error Detection On Reliability Assessment Via Life Testing*, IEEE Transactions on Software Engineering Vol. 20, No. 2, February 1994.

[AUCS96] D. Aucsmith, *Tamper Resistant Software*. Proceedings of the First Information Hiding Workshop. Cambridge, UK

[BANN79] J. P. Banning. *An Efficient way to find the side effects of procedure calls and the aliases of variables.* In Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages. pp29-41, January 1979.

[BLAK79] G. R. Blakley. *Safeguardin Cryptographic Keys.* Proceedings of the National Computer Conference, 1979. Vol 48.

[CER99] C. Cifeuentes, M. Van. Emmerik, N. Ramsey. The Design of a Resourceable and Retargetable Binary Translator. Proceedings of the Sixth Working Conference on Reverse Engineering, Atlanta, USA. October, 1999. pp280-291.

[CHOI91] J. Choi, R. Cytron, J. Ferrante. *Automatic Construction of Sparse Data Flow Evaluation Graphs.* In 18th Annual ACM Symposium on the Principles of Programming Languages. pp55-66.

[CTL97a] C. Collberg, C. Thomborson, D. Low. *Breaking abstractions and unstructuring data structures*, IEEE International Conference on Computer Languages, Chicago, May 1998.

[CTL97b] C. Collberg, C. Thomborson, and D. Low. *A taxonomy of obfuscating transformations.* Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[DEST98] P. Devanbu and S. Stubblebine. *Stack and Queue integrity on hostile platforms.* Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, May 1998.

[DSB96] Report of the Defense Science Board Task Force On Information Warfare -- Defense (IW-D), Office of the Secretary of Defense. November 1996. Available at http://www.jya.com/iwd.htm

[ELKN99] This is the updated version of Matt's tech report.

[EMAM94] M. Emami, R. Ghiya, L. Hendren. *Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers,* In Proceeding of of the SIGPLAN 94 Conference on Programming Language Design and Implementation, pp242-256. SIGPLAN Notices, Vol 29. No. 6. Orlando, Florida USA.

[FGS96] W. Farmer, J. D. Guttman, and V. Swarup. Security for Mobile Agents: Issues and Requirements. In Proceedings of the 19th National Information System Security Conference, Baltimore, Maryland.

[FERR98] A. Ferrari. *Process State Capture and Recovery in High-Performance Heterogeneous Distributed Computing Systems.* Ph.D. Dissertation. University of Virginia, January, 1998.

[FOSO96] S. Forrest, A. Soma. *Building Diverse Computer Systems.* In the Proceeding of the 1996 Hot Topics of Operating Systems Conference.

[GRID97] S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, S. Staniford-Chen, R. Yip, D. Zerkle. *GrIDS: A Graph-Based Intrusion Detection System.* In the Proceeding of the 1997 National Information System and Security Conference, Baltimore, 1997.

[HECH77] M. Hecht. *Flow Analysis of Computer Programs*, Elsevier North-Holland Press, 1977.

[HIND98] M. Hind, A. Pioli, *Assessing the Effects of flow-sensitivity on Pointer Alias Analyses.* Research Report 21251, IBM T. J. Watson Research Center.

[HIND99] M. Hind, M. Burke, P. Carini and J. Choi. *Inter-procedural Pointer Analysis.* ACM Transactions on Programming Languages and Systems, Vol. 21, No. 4, July 1999, pp 848-894.

[IDIP97] D. Schnackenberg. *IDIP concept Document.* Boeing, Personal Communication

[INGE71] F. M. Ingels. *Information and Coding Theory.* Intext Educational Publisher, 1971.

[KEHO97] R. Keller, U. Holzle. *Binary Component Adaptation.* Technical Report, Department of Computer Science, University of California at Santa Barbara, TRCS-97-20.

[KEP97] J. Knight, M. Elder, J. Flynn, P, Summary *of three critical infrastructure systems.* Computer Science Technical Report, CS-97-27, Department of Computer Science, University of Virginia.

[LAMB73] H. Lambert, *System Safety Analysis and Fault Tree Analysis*, Lawrence Livermore Laboratory Report, UCID-16238, 1973

[LAN921] W. Landi. *Undecidability of Static Analysis.* ACM Letters on Programming Languages and Systems, Vol. 1, No. 4. December 1992, pp 323-337.

[LAN922] W. Landi. *Interprocedural Aliasing in the Presence of Pointers.* Ph.D. Dissertation, Rugters University, 1992.

[LARY92] W. Landi, B. Ryder. *A Safe Approximate Algorithm for Interprocedural Pointer Aliasing.* In Proceedings of the 1992 SIGPLAN Symposium on Programming Language Design and Implementation. pp235-248, June 1992.

[LUNT93] T. Lunt. *Detecting Intruders in Computer Systems.* 1993 Conference on Auditing and Computer Technology. 1993.

[MARL93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. D. Choi, M. G. Burke, and P. Carini. *Pointer-induced Aliasing: A clarification.* ACM SIGPLAN Notices, Vol 28, No. 9. pp67-70, September 1993

[MEAD97] C. Meadows. *Detecting attacks on mobile agents.* In Proceedings of the DARPA workshop on Foundations for secure mobile code, Monterey CA, USA, March 1997.

[MUCH97] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[MUNS93] J. C. Munson and T. M. Kohshgoftaar. *Measurement of data structure complexity.* Journal of Systems software, 20:217-225, 1993.

[MYER81] E. Myers. *A Precise Inter-procedural Data Flow Algorithm.* In the conference record of the Eighth Annual ACM Symposium on Principles of Programming Languages. Williamsburg, VA. January, 1981. pp219-230

[NECU00] G. Necula. *Translation Validation for an Optimizing Compiler.* In the proceeding of the ACM SIGPLAN Conference on Programming Language Design and Implementation. May 2000. Vancouver, British Columbia, Canada.

[NEUM00] P. Neumann, *Practical Architectures for Survivable Systems*, Report for the Army Research Lab. 2000.

[PAF] *The Prolangs Analysis Framework.* Rutgers University. http://www.prolangs.rutgers.edu/

[PBCW96] C. Pu, A. Black, C. Cowan, J. Walpole, *A Specialization Toolkit to Increase the Diversity in Operating Systems.* Proceedings: 1996 ICMAS Workshop on Immunity-based systems. Nara, Japan. December 1996.

[PCCP97] Report of The Presidential Commission on Critical Infrastructure Protection, 1997.

[PONE97] P. Porras, P. Neumann, EMERALD: *Event Monitoring Enabling Responses to Anomalous Live Disturbances*. In proceedings of the 1997 National Information Systems Security Conference. Baltimore, 1997.

[PSS98] A. Pnueli, M. Siegel, and El. Singerman. *Translation Validation*. In Bernhard Steffen, editor, Tools and Algorithms for Construction and Analysis of Systems. 4th International Conference, TACAS'98, volume LNCS1384, pp151-166. Springer 1998.

[RISK94] P. Neumann, *Computer-Related Risks*. ACM Press, New York, and Addison-Wesley, Reading, MA, 1994.

[RSA78] R. Rivest, A. Shamir, Adleman. *A method for Obtaining Digital Signatures and Public-Key Cryptosystems,* Communications of the ACM 21,2, February 1978, pp120--126.

[SATS98] T. Sander, C. Tschudin. *Protecting Mobile Agents Against Malicious Hosts*. Lecture Notes of Computer Science, Vol 1419. Mobile Agents. 1998. Springer-Verlag.

[SKDG98] K. Sullivan, J. C. Knight, X. Du, and S. Geist. *Information Survivabiity Control Systems.* Proceedings of the International Conference of Software Engineering, 1998.

[SHAM79] A. Shamir. *How to share a secret?* Communication of the ACM, Vol. 22, Nov. 1979. pp 612-613.

[SUIF] G. Aigner, et al. *The SUIF2 Compiler Infrastructure*. Documentation of the Computer Systems Laboratory, Stanford University.

[SW63] C. E. Shannon, and W. Weaver. *The Mathematical Theory Of Communication* University of Illinois Press, Urbana and Chicago. 1963

[THHA91] M. Theimer, and B. Hayes. *Heterogeneous Process Migration by Recompilation*, Proceedings of the 11th International. Conference on Distributed Computing Systems, Arlington, TX. May 1991

[VGRH81] W. E. Vesely, F. F. Goldverg, N. H. Roberts and D. F. Haasl, "*Fault-tree handbook*". U.S. Nuclear Regulatory Commission Rep. NUREG-0492, 1981.

[YEE97] B. Yee. *A Sanctuary for Mobile Agents*, In Proceedings of the DARPA workshop on Foundations for secure mobile code, Monterey CA, USA, March 1997.