Software Tamper Resistance: Obstructing Static Analysis of Programs

Chenxi Wang, Jonathan Hill, John Knight, Jack Davidson
Department of Computer Science
University of Virginia{cw2e | jch8f | knight | davidson@cs.virginia.edu}

Abstract

In this paper we address the problem of protecting trusted software on untrusted hosts by code obfuscation. We address one aspect of the problem, namely obstructing static analysis of programs.

The presence of aliases has been proven to restrict greatly the precision of static data-flow analysis. Meanwhile, effective alias detection has been shown to be NP-Hard. While this represents a significant hurdle for code optimization, it provides a theoretical basis for structuring tamper-resistant programs—the systematic introduction of nontrivial aliases transforms programs to a form that yields data flow information very slowly and/or with little precision. We describe a set of transformations that introduce aliases and further hinder the analysis by a systematic "break-down" of the program control-flow; transforming high level control transfers to indirect addressing through aliased pointers. By doing so, the basic control-flow analysis is made into a general alias analysis problem, and the data-flow analysis and control-flow analysis are made co-dependent.

We present a theoretical result which shows that a precise analysis of the transformed program, in the general case, is NP-hard and demonstrate the applicability of our techniques with empirical results.

1. Introduction

Protection of information against unauthorized access is a central issue in computer security. In this paper we consider a special case of this general problem, the protection of an executable program in an insecure environment. In a more formal sense, we wish to execute software in an untrustworthy environment and have some level of assurance that the program will execute as expected.

This is precisely the opposite of the malicious mobile-code problem in which one wishes to protect a trusted system against software that is not necessarily trustworthy because it arrives over a network.

Software protection is important in computer security; it arises, for example, in intrusion-detection systems. The parts of the intrusion-detection system that record events have to be trusted (or we could not trust the intrusion-detection results) yet these parts often operate on hosts that might have been penetrated.

More generally, the problem is central to the creation of survivable network systems since any mechanism designed to assist a network in detecting and recovering from traumas is a tempting target for intruders [8]. In fact, the problem arises in any

environment where remote execution takes place[1]. In an online game environment, for example, a player might try to compromise the game by manipulating the data or code on his client machine. Clearly in this case we wish to be able to trust the game software when it is operating on an untrustworthy host.

The problem of software protection cannot be addressed using traditional security means. For example, access control is infeasible since it must be carried out by the host that cannot be trusted. Similarly, the software we seek to protect cannot be encrypted because it is executable software[2] and it has to be in an executable form ready for the processor. In principle, the program could be decrypted incrementally under software control but the impact on performance would be unacceptable and, in fact, would not necessarily be secure if the decryption process takes place on the same host.

What is needed, in this case, is tamper resistant software [2]. This paper addresses one aspect of software tamper resistance—prevention of static analysis of programs. Our premise is that intelligent tampering attacks require knowledge of the program semantics, and this knowledge may be acquired through static analysis. In this paper, we introduce a compiler-based approach to perform code transformations that are designed to obstruct static analysis. The key difference between our approach and previously proposed code-obfuscation techniques [3,6,4] is that our techniques are supported by both theoretical and empirical complexity measures. Without the complexity measures, code-obfuscation techniques are at best ad hoc.

The structure of the paper is as follows: In section 2, we present the system model and assumptions on which this work is based. Section 3 describes the basics of static analysis. Section 4 and 5 present the transformations to hinder control-flow and data-flow analysis. Section 6 and 7 discuss theoretical and practical foundations of the proposed scheme. Section 8 and 9 present our implementation and experimental results.

2. System Model and Assumptions

Before we delve into the solution approach, we first describe the assumptions and the

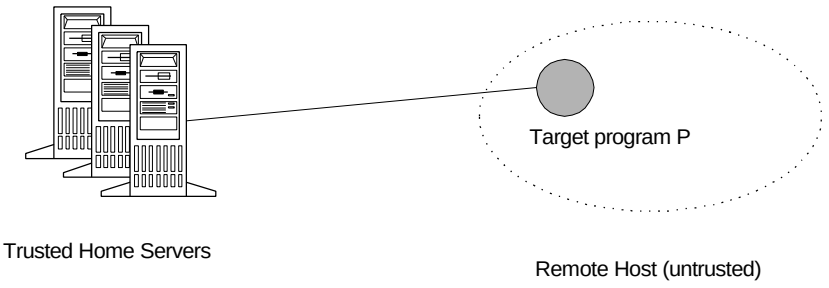


Figure 1: A remote execution model

[1] The degree of severity varies depending upon the application context.

[2] A special case exception exists, see 16 for details.

system model on which the solution is based. The system model is depicted in Figure 1.

Several characteristics and assumptions about this model are important to our solution approach:

First, in our system, the program we seek to protect is dispatched from a (or multiple) *trusted* home server. While executing on the remote host, the program *P* communicates with the home server in a prescribed fashion (e.g., via predetermined protocols).

Second, we assume trusted network communications. We assume the use of cryptographic protection and secure communications protocols for network communications – attacks such as traffic replaying, insertion, and eavesdropping are assumed impossible[3].

Third, in this work, we are concerned with programs that, while executing remotely, maintain a high level of interaction with their home server. In other words, a computation-intensive application with very little or no interaction with its home is not the model of computing considered in this work.

Lastly, this work is primarily interested in defending against 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. It may result in a denial of execution however, since the tampered program or data can cause execution to fail.

- **Impersonation.** An impersonation attack is similar to intelligent tampering in that the attacker seeks to establish a rogue version of the legitimate program. The difference lies in that the former attempts to emulate the behavior of the original program, while the latter aims to modify the program or its data directly.

The first three assumptions allow the target software to incorporate some self-checking mechanisms whose results can be verified by the trusted home server[4].

---

[3] Denial-of-service attacks are still possible, but they can be readily identified.

[4] It is not the objective of this paper to devise the checking mechanisms. We simply assume such mechanisms exist.

The last assumption states that attacks such as explicit denial-of-execution are not considered in this work. With the checking mechanism in place, denial-of-execution produces straightforward symptoms that can be readily identified by the home server (e.g. loss of communication). Unlike denial-of-execution, an intelligent tampering or impersonation attack is not always immediately obvious; if the attacker has detailed knowledge of what the software is supposed to do and the appropriate privilege to instantiate a malicious copy, he can replace the original program and make the replacement virtually undetectable. Such attacks therefore have the potential to inflict substantial harm—the adversary could manipulate the program to perform seemingly valid but malicious tasks.

In order to mount a successful intelligent tampering or impersonation attack, the adversary must first defeat the checking mechanism. This requires information about the program semantics and the behavior of the checking mechanism, and it is this information that we endeavor to protect. For example, consider the following code segment:

```
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, he or she needs to understand whether and how the *Check_for_intrusion()* function changes the values of *a* and *b*. Without this knowledge, his 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 action, and this boils down to a understanding of the program semantics. One way this understanding can be acquired is through program analysis. Program analysis can be conducted either statically, or dynamically on a trace of the execution. This paper focuses on obstruction of static analysis. Our approach consists of a set of code transformations designed to increase the difficulty of such analyses.

3. Static Analysis of Programs

Static analysis refers to techniques designed to extract information from a static image of a computer program. Static analysis is more efficient than analyses performed dynamically such as tracing of an execution. Traditionally, static analyses are often used to gather information on the "modification, preservation and usage of data quantities" for the purpose of code optimization [7].

From the software-protection point of view, static analysis could yield useful information for targeted manipulation of software. Consider again the code example in the last section. A *use-def* analysis [9] of the code segment would quickly reveal that a possible

definition of the data quantity *a* in function *Check_for_intrusion()* will be propagated to a use (through its alias *p*) in function *integrity_check()*. Based on this knowledge, an adversary could then perform specific modification to *Check_for_intrusion()* so long as he leaves the semantics of *a* intact for its later use.

Static analysis can be conducted in a manner that is either sensitive or insensitive to the program control-flow. Flow-insensitive analysis is generally more efficient at the price of being less precise [9]. In the context of the problems we are considering, flow-insensitive analyses will not yield information that is sufficient for intelligent tampering. Therefore we will consider flow-sensitive analyses only.

A control-flow sensitive analysis entails the following essential steps:

a) Conduct control-flow analysis to build the flow graph of the program. A flow graph consists of nodes which are basic blocks, and edges that indicate control transfers between blocks. This step provides information that is essential in various data-flow analyses.

b) Construct the problem of gathering target information as a data-flow problem and conduct data-flow analysis on the flow graph.

It is important to note that control-flow analysis is the first stage of the analysis—it provides information on the program call structure and control transfer that is essential for subsequent data-flow analysis. Without this information, 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 technical basis of our approach to defeating static analysis is to make the program control-flow *data-dependent*. That is, control-flow and data-flow analysis are made co-dependent. The results of this co-dependence are: (1) vastly increased complexity of both analyses; and (2) reduced analysis precision.

4. Control-flow Transformations

Determining the basic program flow graph is a straightforward operation when branch instructions and targets are easily identifiable—it is a linear operation of complexity *O(n)*, where *n* is the number of basic blocks in the program.

Real-world programs tend to have control-flow that can be easily discerned, as this is encouraged for program clarity and enforced by high-level language constructs. This flow information is utilized in flow-sensitive analysis to attain better analysis resolution.

The first set of code transformations that we employ modify high-level control transfers to obstruct static detection of branch targets and call destinations. We perform this transformation in two steps. In the first step, high-level control structures are decomposed into *if-then-goto* constructs. This transform is illustrated in Figure 2 in which the sample program in Figure 2.a is transformed into the structure in Figure 2.b.

Secondly, we modify the *goto* statements such that the target addresses of the *goto*'s are determined dynamically. This is done by loading from the content of some data variable location instead of a direct jump address. In *C*, we implement this by replacing the *goto* statements with an entry to a *switch* statement, and the switch variable is computed in each code block to determine which block is to be executed next. The transformed code is depicted in Figure 3.

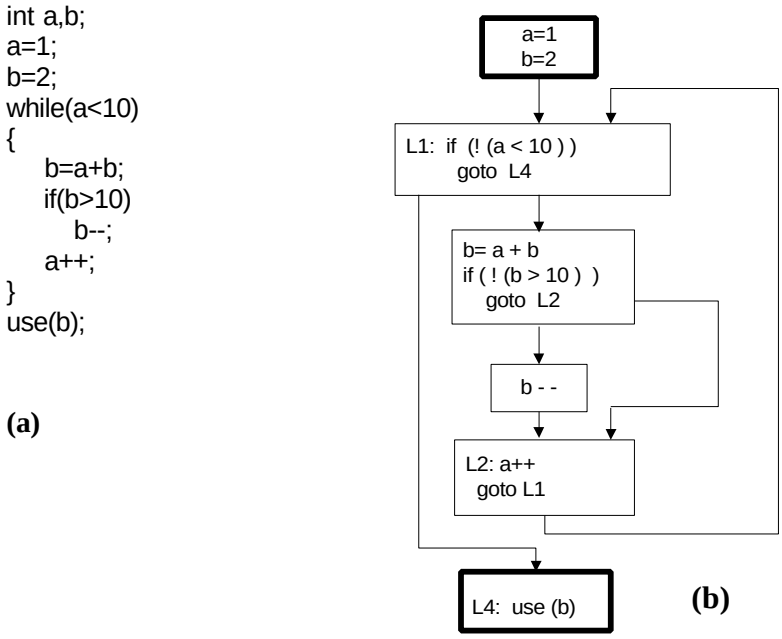


Figure 2: Dismantling of high-level constructs

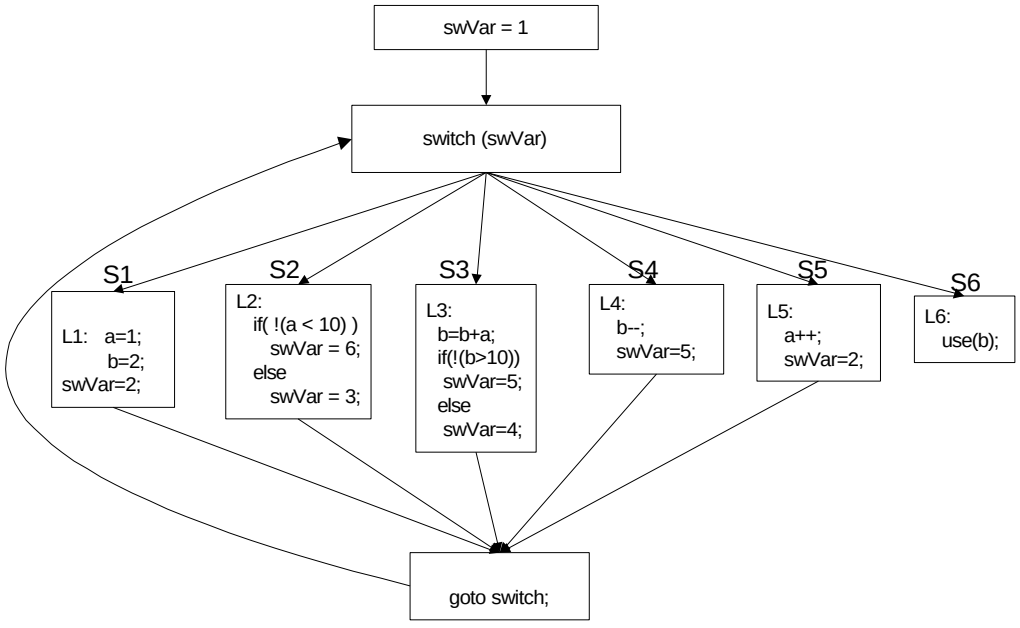


Figure 3: Transform to indirect control transfers

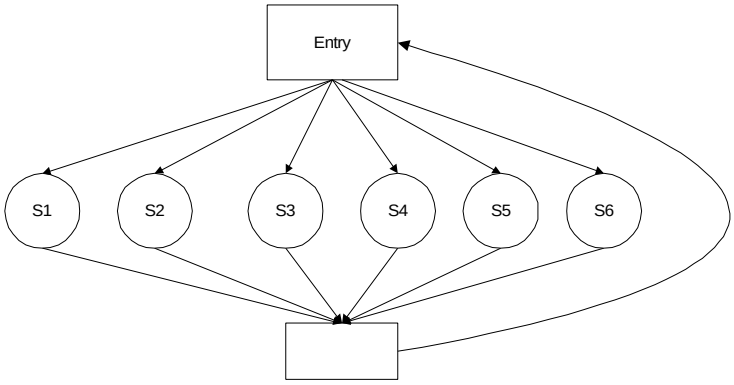


Figure 4: A flattened control-flow graph

With the above transformations, direct branches are replaced with data-dependent instructions. As a result, the flow graph that can be obtained from static branch targets degenerates to a common form shown in Figure 4. We will informally refer to a degenerated flow graph as 'flattened'. It can be shown that this degenerate form is equivalent to the control-flow perceived by a flow-insensitive analysis [11]. Without knowledge of the branch targets and the execution order of the code blocks, every block is potentially the immediate predecessor of every other block.

Observe that the placement of the basic blocks does not need to follow the general order of execution; the blocks can appear in a completely random order (with the exception of the entry block). Also note that artificial code and branch instructions can be introduced so that sizes of the code blocks can be arbitrarily small or large (within certain bounds).

Without branch-target information, the complexity of building the flow graph statically is determined by how easy it is, at each branching point, to discern the latest definition of the switch variable. This is exactly a classical *use-and-def* data-flow problem [7]. The complexity of data-flow analyses is influenced by various program characteristics such as aliasing [9, 12]. We will show in the next section how manipulation of certain program characteristics can yield additional complexity for data-flow analysis and ultimately render static analysis a difficult, if not entirely infeasible, problem.

5. Data-flow transformations

After the control-flow transformations described in Section 4, the complexity of building the program flow graph now hinges on the complexity of determining branch targets, which is in essence a *use-def* data-flow problem.

Many classical data-flow problems are proven to be NP complete or harder (some are known to be undecidable) [10, 13]. A fundamental difficulty that data-flow analysis must deal with is the existence of aliases in the program. Aliases happen when two or more

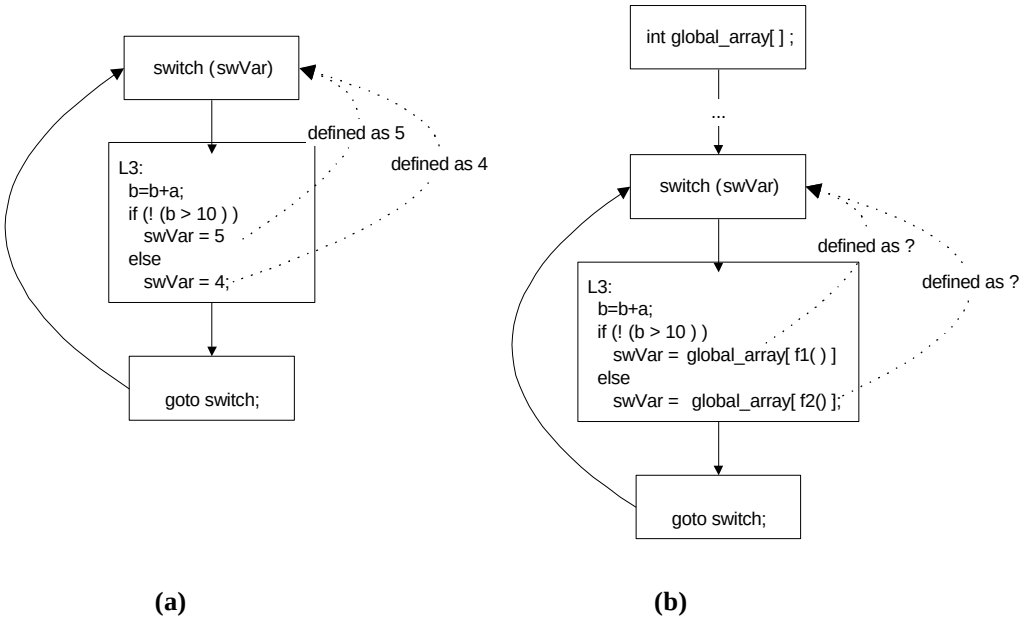


Figure 5: Example illustrating dynamic computation of the switch variable

names refer to the same memory location. Alias detection is essential to many data-flow analyses. For example, in order to precisely determine the live definition problem, a data-flow algorithm must understand the alias relationships among variables since data quantities can be modified when assignments are performed on any aliased names.

Precise alias detection in the presence of general pointers and recursive data structures is known to be undecidable [13], 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 include the following techniques:

Dynamic computation of branch targets: Consider the code segment in Figure 5.a. A *use-def* analysis to analyze where the switch variable *swVar* (contains branch target information) is defined is straightforward (the dashed line indicates an *use-def* information chain). Now consider the code segment in Figure 5.b in which a global array "*global_array*" is introduced and the value of *swVar* is computed through the elements of the array (*f1()* and *f2()* indicate complex expressions of subscript calculation). Replacing the constant assignment in Figure 5.a with indirect accesses of the array implies that the static analyzer must deduce the array values before the value of *swVar* can be determined.

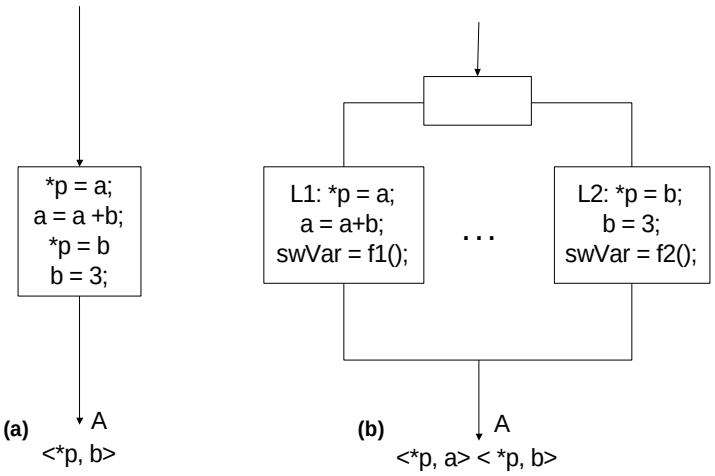


Figure 6: Introducing aliases through pointers

Aliases through pointer manipulation: We introduce aliases in the following steps:

- In each function, we introduce an arbitrary number of pointer variables
- We then insert artificial basic blocks, or code in existing blocks, that assign the pointers to data variables including elements of the global array.
- 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 definition 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.

For example, a static analysis performed on the code segment in Figure 6.a can easily discern that only the second definition of the pointer variable *p* will carry to point A in the program. However, if the basic block in Figure 6.a is decomposed into two blocks and the transition between blocks is obfuscated using our *flatten-and-jump* technique as depicted in Figure 6.b, the static analyzer will report both alias relations <*p, a> and <*p, b> since it does not know which block executes first.

Figure 7 illustrates example transformations as applied to the program in Figure 2.a. The result of the transforms is the following: a static analyzer will report imprecise alias

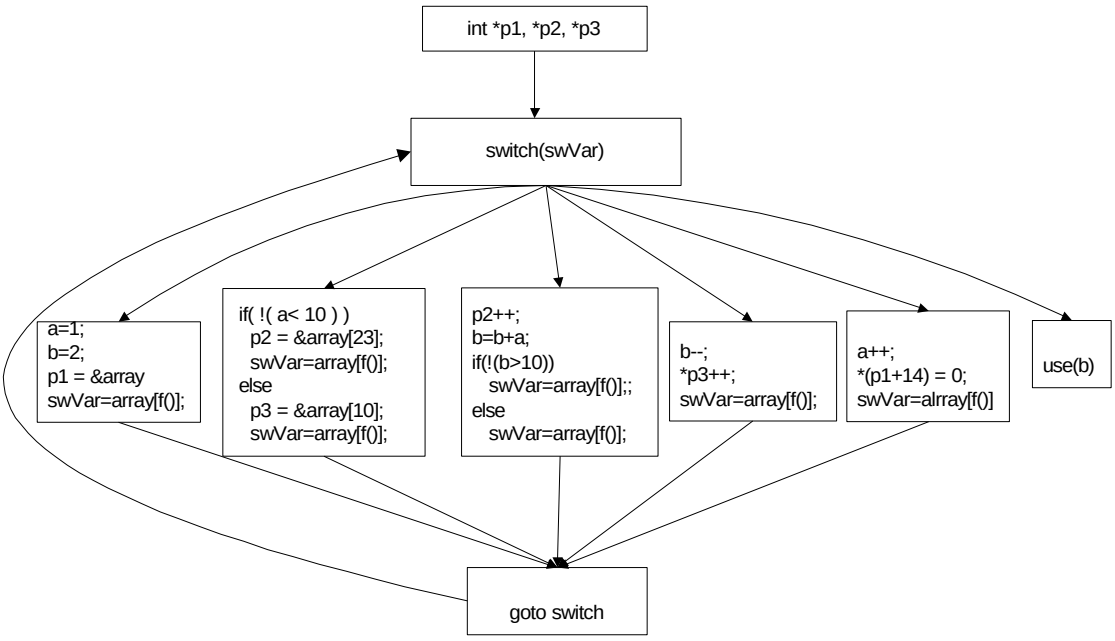


Figure 7: Example transform using pointer manipulation

relations that suggest that the global array is altered, and that its contents do not remain static. With sufficient alias introduction, the analysis will resolve an array element to a large set of possible values. This in turn implies that, at each use, the switch variable can take on a large set of values.

It can be argued that if an adversary can capture the initial value of *swVar*, he can then find the first block to be executed, and from there the next block can be identified. Following this he might be able to recover some of the original control-flow. While this requires simulation of the code in a block in order to identify the next one, the simulation only needs to be done once for each block. As a result this technique lies somewhere between static analysis and a full execution trace, with analysis time being proportional to the number of blocks in the program.

One way to defeat the above technique is by unrolling loops and introducing semantically equivalent blocks that will be chosen randomly during execution. The result is that the complexity of recovering the program control-flow will be comparable to an effective simulation. Additionally, the initial computation of *swVar* can be erased from memory once it is used to avoid unnecessary exposure of information.

6. Complexity Evaluation

We have thus far conjectured that the difficulty of discerning indirect branch target addresses is influenced by aliases in the program. In this section, we support this claim by

presenting a proof in which we show that statically determining precise indirect branch addresses is a 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-hard.

Proof: Our proof consists of a polynomial time reduction from the 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 [10]. Landi later proposed a similar proof to prove that alias detection is NP-complete in the presence of general pointers [12].

Consider the 3-SAT problem for $\wedge_{i=1}^{n}(V_{i1} \vee V_{i2} \vee V_{i3})$ 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 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 $\{v_2 = \&true; \overline{v_2} = \&false\}$

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:

Code segment L1 declares the variables and an array A[]. $v_1, v_2, ...v_m$ are doubly dereferenced pointer variables. L2 indicates A[*true] points to the address of function 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 $<*v_i, true>$, 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, 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

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

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 branch target addresses.

7. Practical Complexity Evaluation

While the theoretical results in the last section bode well for the alias-based code transformations, we still need to evaluate our approach against possible heuristics and approximation methods. In this section, we explore the effect of two heuristics: brute-force search and alias approximations.

7.1. Brute-force search method

To determine the execution order of the program blocks, an adversary might employ a brute-force search method in which all combinations of the code block ordering are explored. This is a naive exhaustive search heuristic in which each block is considered equally likely to be the immediate successor of the current block (including the current block itself). The time complexity of such a brute-force method is $O(n^k)$, where n is the number of distinct program blocks and k is the number of blocks in an execution chain (k is equivalent to the number of times the *switch* loop in Figure 4 is traversed). Clearly, this represents the worst-case time complexity and is extremely inefficient when the value of n and k are sufficiently large.

7.2. Alias-detection approximation methods

The problem of precise alias detection in the presence of general pointers and recursive data structures is undecidable [13]. In practice, therefore, imprecise and safe algorithms are often used for alias detection [11, 13]. A safe approximation algorithm reports a super set of the actual alias relationship—a trivial case is to assume every variable is aliased to every other variable in the current scope.

In this section, we will demonstrate that through careful application of our transforms, state-of-the-art alias analysis algorithms are not sufficient to recover the original control flow of the program. To this end we will utilize the NPIC pointer alias analysis algorithm by Hind, Burke, Carini and Choi [11]. Their algorithm is both interprocedural and flow sensitive, and has been shown to be highly effective on the SPEC benchmark programs.

We use a test program that is a *C* implementation of the *8queens* problem[5]. Our goal is to determine whether an adversary utilizing the NPIC algorithm can recover the original control flow from the transformed program.

We first perform the following code transformations on the *8queens* program.

- First, the program control-flow is degenerated as described in section 4, and a global array—"*global _array*"—is introduced.
- Second, pointers are introduced that are aliased to meaningful data as well as the *global_array* elements.
- Lastly, we insert code and code blocks to manipulate pointers in the following way:
    - Assignments to the pointer variables are created
    - Program data values are manipulated through the pointers

  Of all the artificial code blocks introduced, only the first block is ever executed, and the computation in this block represents the correct assignment to maintain the semantics of the program.

Our next step is to apply the analysis algorithm and derive *use-def* information on the *swVar* assignments. This was done by hand as the implementation of the algorithm has not been made available[6]. The step-by-step analysis is too voluminous to be included in this paper. In a nutshell, the algorithm terminates and reports that the elements of the *global_array* have possibly changed arbitrary number of times and thus the array elements contain arbitrary values. Since the switch variable computation is done through the array elements, its value cannot be determined statically.

8. Implementation

This section describes the implementation of our approach. Our current implementation is limited to source-to-source transformations for the *C* programming language. However, most techniques described in the paper are not restricted to *C* and can be extended to other languages with a similar pointer paradigm.

---

[5] The 8queens problem is a recursive search problem.

[6] The implementation of the NPIC algorithm is no longer maintained and distributed.

In our implementation, we use the SUIF2 system to develop compiler passes to implement the code transformations [1]. The SUIF2 system provides tools for program manipulation in the SUIF representation, and transform utilities are provided between the SUIF representation and the source language.

Each compiler pass traverses the SUIF representation and performs the desired modifications. The exact code modifications are determined by a random seed: that is, the resulting program is different for each compilation. For example, the layout of the global array, the exact percentage of the control-transfers that are transformed, and the number of no-op code blocks that are added are all determined by a random number generated from the seed.

In addition, the random seed determines the layout of the program blocks. That is, in the final machine representation, the code blocks are randomly placed. This random placement, albeit by itself does not adding any additional security, represents a cheap obfuscation that will come handy in defeating naive pattern-matching algorithms.

The randomization in the transformation allows generation of diverse versions of the original source program. Variations between the different versions facilitate the notion of installation-unique software. If we can institute the variations in such a way that analysis must be specially tailored for each installation and a new installation is made to occur at random intervals, resources required to mount an attack would rise significantly. More detailed discussions of this installation-specific scheme can be found elsewhere [17].

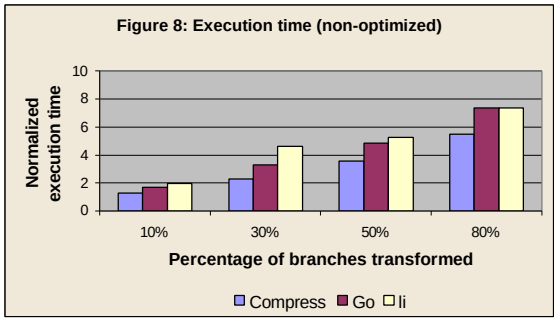
9. Performance Results and Empirical Evaluations

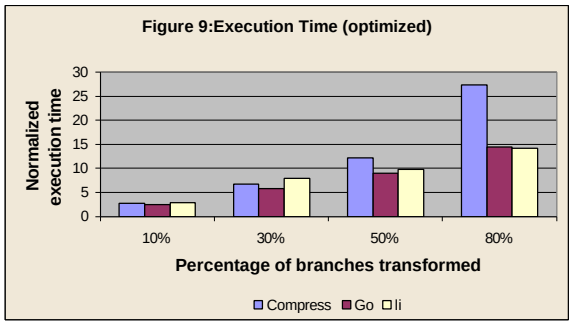
This section reports performance results obtained with experimental transformations on the SPEC95 benchmark programs. Of issue here are three measures: *Performance of the transformed program, performance of static analysis, and precision of static analysis*.

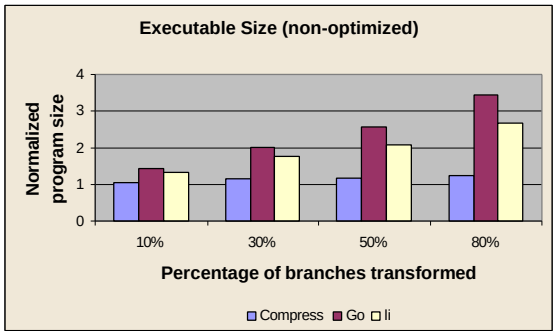
By performance of the transformed programs, we mean the execution time and the executable object size after transformation. These measures reflect the cost of the transformation. By performance of static analysis, we mean the time taken for the analysis tool to reach closure and terminate. A related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

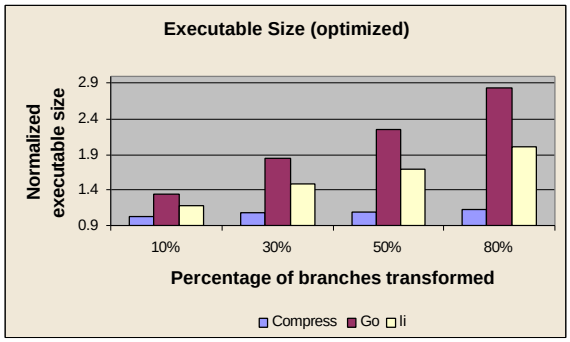
9.1. Performance of the transformed program

The following data is obtained by applying our transforms to SPEC95 benchmark programs. Three SPEC programs are 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.









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. The experimental results show that, in both cases, the performance-slowdown increases exponentially with the percentage of transformed branches in the program. On average, the 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 *Compress*—a whopping 350% decrease in the execution time due to optimization. However, as can be seen in Figure 9, our transform removed significant optimization potential from *Compress;* the execution speed of the transformed and optimized *Compress* diverges most significantly from the performance of the original optimized program. As *Compress* 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. *Go*, a branch-intensive program, shows 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 *Compress*. *Compress* contains relatively fewer static branches, resulting in less potential for code growth with the transform.

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 future 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.

9.2. 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 [11] and the PAF toolkit from Rutgers university [15]. 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 [13]. Both NPIC and PAF perform control-flow analysis exactly once with no further refinement on the flow graph[7].

In our experiments, PAF successfully analyzed small, toy programs but failed to handle some of the large programs (before transformation) included in the SPEC benchmarks. We tested PAF on a wide range of toy programs that contain extensive looping constructs and branching statements. In all of the test cases, PAF terminates 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.

10. Conclusion

Software tamper resistance, the problem of protecting software from untrustworthy hosts, is important in many network applications. In this paper, we considered one significant class of attacks, namely those based on static analysis of the binary form of the software. We have presented a strategy for defeating analysis by making the control-flow analysis of the program dependent on the data in the program. Since data-flow analysis of acceptable precision must itself be dependent on the control-flow information, this

[7] NPIC performs refinement on the procedure call graph but not the intra-procedural flow graph.

approach is capable of expanding analysis times considerably and reducing the precision of the analysis to useless levels. The theoretical bound that we have established shows that analysis of programs that have been transformed in this manner is NP complete.

The practical instantiation of the transformation we have developed makes a number of changes to the source code of a program including flattening the control-flow graph, and introducing aliases and data-dependent branches. In experiments that we have conducted on sample programs, the transformed versions defeat currently available static-analysis tools. Although such experiments are not and could never be definitive evidence, we regard these results as promising indications that we have a practical approach to defeat static analysis.

Our future work includes exploring the possibility of extending the current scheme to an inter-procedural level with the use of function pointers and aliases. We will also investigate establishing the practical limits of the transformation technology. We note that once a practical lower bound on the time needed to analyze a transformed program is established, it will be possible to protect systems for extended times by periodic replacement of the target program. Randomization within the transformation will ensure that analysis of one transformed version of the program will not be useful in the analysis of other versions.

11. References

1 G. Aigner, *et al*. "The SUIF2 Compiler Infrastructure". *Documentation of the Computer Systems Laboratory*, Stanford University.

2 D. Aucsmith. "Tamper Resistant Software". *Proceeding of the 1st information hiding workshop*, Cambridge, England, 1996.

3 C. Collberg, C. Thomborson, D. Low. "Breaking Abstractions and Unstructuring Data Structures", *IEEE International Conference on Computer Languages*, Chicago, May 1998.

4 C. Collberg, C. Thomborson, and D. Low. "A Taxonomy of Obfuscating Transformations". *Technical Report 148*, Department of Computer Science, University of Auckland, July 1997.

5 S. Forrest, A Soma. "Building Diverse Computer Systems". In the 1996 *Proceedings of the Hot Topics of Operating Systems.*

6 F. Hohl. "Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts". In *Lecture Notes in Computer Science, vol. 1419, Mobile Agents and Security*. Edited by G. Vigna. Springer-Verlag, 1998.

7 M. Hecht. "Flow Analysis of Computer Programs". Elsevier North-Holland, New York. 1977.

8 J. Knight, K. Sullivan, M. Elder, C. Wang. "Survivability Architectures: Issues and Approaches" In *Proceedings: DARPA Information Survivability Conference and Exposition*. IEEE Computer Society Press. Los Alamitos, CA, January 2000, pp. 157-171.

9 S. Muchnick. "Advanced Compiler Design Implementation". Morgan Kaufmann, 1997.

10 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.

11 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.

12 W. Landi. “Interprocedural Aliasing in the Presence of Pointers”. Ph.D. Dissertation, Rugters University, 1992.

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

14 W. Landi, B. Ryders. “A Safe Approximation Algorithm for Interprocedural Pointer Analysis”. *CS Technical Report*, Rutgers University, 1991.

15 The Prolangs Analysis Framwork (PAF). Rutgers University. http://www.prolangs.rutgers.edu/public

16 T. Sander, C. Tschudin. “Protecting Mobile Agents Against Malicious Hosts”. In the *Proceedings of the 1998 IEEE Symposium of Research in Security and Privacy*. Oakland, 1998.

17 C. Wang, J. Knight, “Running Trusted Code on Untrustworthy Platforms”. Computer Science Technical Report, CS-99-18. University of Virginia.