Protection of Software-based Survivability Mechanisms

Chenxi Wang, Jack Davidson, Jonathan Hill, John Knight*

Department of Computer Science
University of Virginia
{cw2e | jch8f | knight}@cs.virginia.edu

Microsoft Research*
Redmond, WA
jwd@microsoft.com

Abstract

Many existing survivability mechanisms rely on software-based system monitoring and control. Some of the software resides on application hosts that are not necessarily trustworthy. The integrity of these software components is therefore essential to the reliability and trustworthiness of the survivability scheme. In this paper we address the problem of protecting trusted software on untrustworthy hosts by software transformations. Our techniques include a systematic introduction of aliases in combination with a "break-down" of the program control-flow; transforming high-level control transfers to indirect addressing through aliased pointers. In so doing, we transform programs to a form that yields data flow information very slowly and/or with little precision. 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

In building survivable systems, many existing mechanisms [8, 9] rely on software-based network monitoring and management. Because some of the software components for the survivability mechanism will execute on hosts that are not necessarily trusted, the reliability and trustworthiness of the survivability mechanism is, therefore, of a great concern.

In this paper, we address the problem of software protection in a potentially malicious environment. We study this problem within the context of a survivable distributed system [9]. In this system, software probes are deployed onto network nodes for monitoring and control purposes. These probes are dispatched from a set of trusted servers. Each probe may employ different algorithms for monitoring local information and for communication with the servers. For instance, different probes might use different data sequences, transmit with a different protocol, or monitor different information. To defeat this network-wide monitoring mechanism, and thereby obtaining control of the network, an adversary must deduce either the algorithm that the probe uses when monitoring or the protocol with which the probe communicates with the server. Each of these attacks requires some level of understanding of the program behavior, which can be obtained through program analysis. This paper addresses one important aspect of software protection—prevention of static analysis of programs.

Static program analysis can reveal a great deal of information about the program such as the control flow and possible uses of data quantities at run-time [11]. This information can be used to facilitate dynamic analysis of the program, and in some cases, aid direct tampering with the program. In this paper, we introduce a compiler-based approach to harden software against static analysis. The basic approach consists of a set of code transformations that are designed to obstruct static analysis. The key difference between our approach and previously proposed code-obfuscation techniques [4, 5, 7] 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 problem of software protection has been investigated in other studies. The notable ones include INTEL's IVK project [2], Collburg's code obfuscation work [4, 5] and mobile cryptography [20]. The IVK work coined the phrase *Tamper Resistant Software*. Their technique was novel but came with the price of considerable run-time cost. The mobile cryptography study proposed a technique to execute programs in an encrypted form. In its present form, the technique has limited applicability (e.g., rational functions).

The approach described in this paper is developed based on well-understood programming language principles, which serve as the basis for the complexity measures. We structure the paper 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. Sections 4 and 5 present the transformations to hinder control-flow and data-flow analysis. Sections 6 discusses theoretical and practical foundations of the proposed scheme. Sections 7 presents our implementation and experimental results.

2. The system model

In this section we describe the assumptions and the system model to set the context for discussion. Our system consists of a set of computing hosts connected via a network and a set of communicating processes running on these hosts. The hosts are divided into two categories: application hosts and survivability control hosts. The processes relevant to the survivability control mission are the *control processes* running on the control hosts and the *probe programs* running on the application hosts. The probes are responsible for local monitoring and reconfiguration. The control processes collect monitoring information from the probes, conduct network-wide analysis, and issue reconfiguration commands to the probes if real-time changes are deemed necessary. An overview of the system architecture is depicted in Figure 1.

Several characteristics and assumptions about the system are important for the discussion. They are listed below:

- **Trusted control servers:** In our system, the control servers and the control processes running on top of them are presumed trusted.

- **Trusted network communications:** We assume the network communications between the control processes and the software probes are trusted.

- **Diversity in the probing mechanism:** In this system, the probing mechanism makes use of two forms of diversity that are essential to the approach detailed in later sections. They are *temporal diversity* and *spatial diversity*. Temporal diversity takes the form of periodic replacement of the probes with a new version dispatched from the trusted control servers. Spatial diversity refers to the installation of different versions of probes across the network. Each version of the probes may use a different probing algorithm, a different protocol to communicate with the control server, and may appear to have a different operational semantics [21]. The use of diversity here makes it essential for an adversary to learn the program algorithm in order to launch an intelligent tampering or impersonation attack.

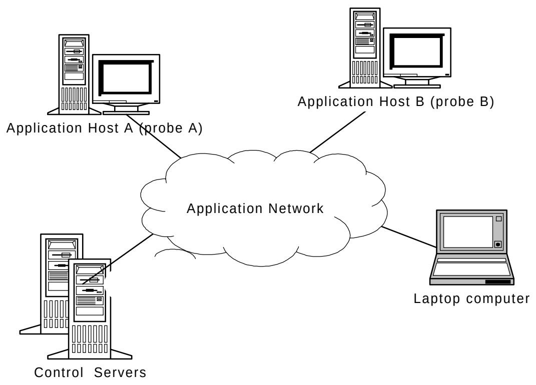


Figure 1: A controlled network

- **High level of interactions between the probes and the control processes:** While executing on a remote host, the probes maintain a high level of interaction with its control server via predetermined protocols. It is assumed that the probes perform integrity checks whose results are verified by the control servers. The checking mechanisms are also installation-unique in that each probe program may employ a set of different checks. It is not the task of this paper to devise the checking mechanisms. It suffices to state that the integrity checks may be performed on the software itself as well as its executing environment. The result of the integrity checks establishes the basis of the probe's identity and authenticity.

In this work, we are primarily interested in defending against sophisticated attacks that fall under 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.

- **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.

It should be noted that denial-of-execution attacks are not considered here. In this problem context, denial-of-execution produces straightforward symptoms that can be readily identified by the trusted server (e.g. loss of communication). Unlike denial-of-execution, an intelligent tampering or impersonation attack may not be 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.

With the diversity scheme and the integrity checks in store, a successful intelligent tampering or impersonation attack requires knowledge about the probe algorithm and the communication protocol in order to bypass or defeat the checking mechanism. This in turn requires information about the program semantics, 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*, and whether *a* or *b* will be used later in the program. Without this knowledge, his action can be revealed when *integrity_check(p)* is called.

Our premise is that an adversary aiming to tamper with or impersonate the program in an intelligent way must understand the effect of his action, and this boils down to an understanding of the program semantics. One way this understanding can be acquired is through program analysis. This paper deals with obstruction of program analysis, in particular, static analysis of programs. Our approach consists of a framework of code transformations designed to increase the difficulty of static program analysis, and that is described in the remaining sections.

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 often more efficient than analyses performed dynamically such as simulated execution.

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 [11] 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 [11, 14].

A flow sensitive analysis first constructs the Control-Flow Graph (CFG) of the program. Such a graph consists of nodes which are basic blocks and edges that indicate control transfers between blocks. The analysis then proceeds to solve the data-flow problem based on the CFG.

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 transform the program control-flow to a highly *data-dependent* nature; that is, the control-flow and data-flow analysis are made co-dependent. The results of this co-dependence are: (1) increased complexity of both analyses; and (2) reduced analysis precision.

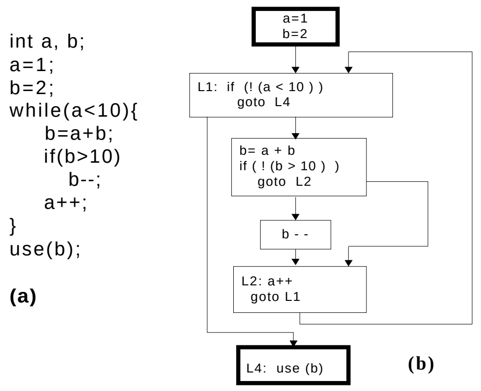
4. Degeneration of control-flow

In a normal program, determining the CFG 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.

The first set of code transformations that we employ modify high-level control transfers to obstruct static detection of the program CFG. We perform this transformation in two steps. In the first step, high-level control structures are transformed into equivalent *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 *goto* target addresses are determined dynamically. In C, we implement this by replacing the *goto* statements with an entry to a *switch* statement, and the switch variable is computed dynamically to determine which block is to be executed next. The transformed code (based on the code segment of Figure 2(a)) is depicted in Figure 3.

With the above transformations, direct branches are replaced with data-dependent instructions. As a result, the CFG that can be obtained from static branch targets degenerates to a flattened form shown in Figure 3. It can be shown that this degenerate form is equivalent to the control-flow perceived by a flow insensitive analysis [14]. Without



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

Figure 2: Dismantling high-level constructs

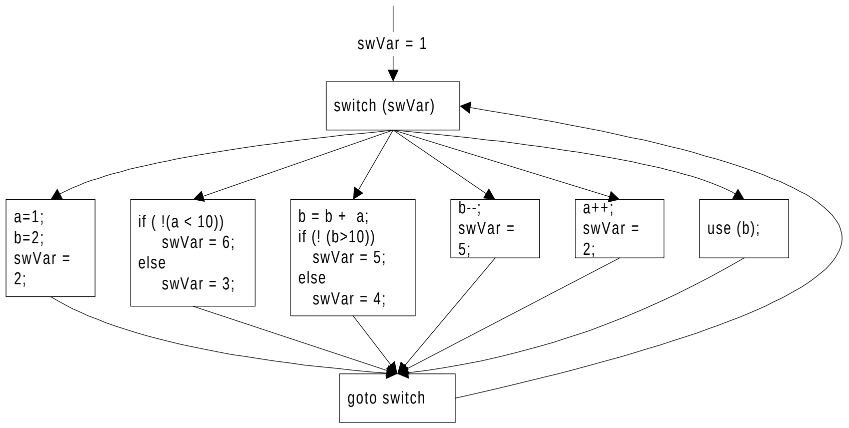


Figure 3: Transforming to indirect control transfers

knowledge of the branch targets and the execution order of the code blocks, every block is potentially the immediate predecessor and/or successor of every other block.

In the absence of the branch-target information, the complexity of building the static CFG 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-n-def* data-flow problem [11]. Note that we have transformed the control-flow analysis into a data-flow problem. The complexity of data-flow analyses is influenced by various program characteristics such as aliasing [11]. We show in the next section how manipulation of data flow characteristics can yield additional complexity for data-flow analysis and ultimately render static analysis an extremely difficult problem, if not entirely infeasible.

5. Data-flow transformations

After the transformations described in Section 4, the complexity of building the CFG now hinges on the complexity of determining branch targets, which is in essence a *use-n-def* data-flow problem. Many classical data-flow problems are proven to be NP-complete[12, 16]. A fundamental difficulty that data-flow analysis must deal with is the existence of aliases in the program. Alias detection is essential to many data-flow problems. For example, in order to 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.

Our second set of transformations focuses on the introduction of non-trivial aliases into the program to influence the computation and the analysis of the branch targets. These transformations include the following techniques:

Index computation of branch targets: Consider the code segment in Figure 4(a). A *use-n-def* analysis to analyze where the switch variable *swVar* (contains branch target information) is defined is straightforward (the dashed line indicates a *use-def* information chain). Now consider the code segment in Figure 4(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 4(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

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

- In each function, we introduce an arbitrary number of pointer variables. We insert artificial basic blocks, or code in existing blocks, that assign the pointers to existing data variables including elements of the global array.
- We replace references to variables and array elements with indirection through these pointers. Previously meaningful computations on data quantities are replaced with semantically equivalent computation through their aliased names (assignments to the *global_array* elements may appear as assignments to a pointer variable)
- As much as possible, uses of the pointers and their definitions are placed in different blocks. This is to introduce difficulties for the *use-n-def* analysis.

Some of the basic blocks will execute, 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 blocks, the analyzer will not be able to deduce which definition is live at each use of the pointer—all pointer assignments will appear live.

For example, a static analysis performed on the code segment in Figure 5(a) can quickly determine that only the second definition of the pointer variable *p* will carry to point A during execution. However, if the basic block in Figure 5(a) is decomposed into two blocks and the transi-

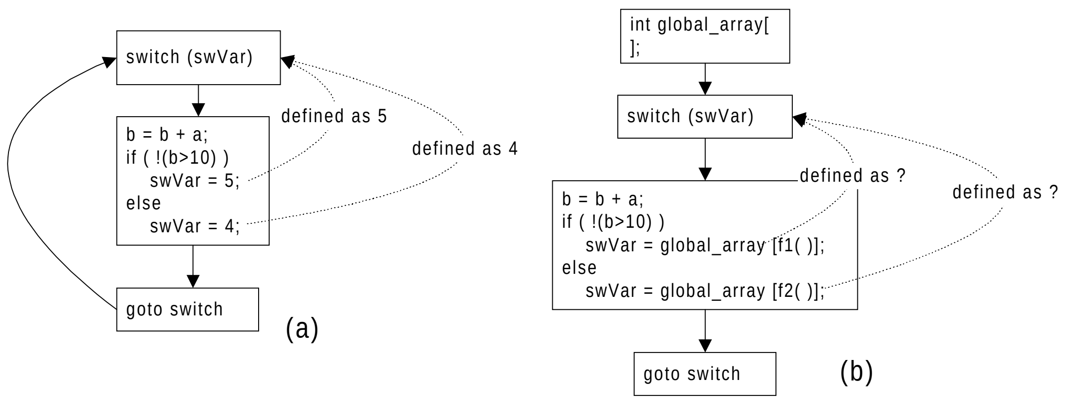


Figure 4: Example illustrating dynamic computation of branch targets

tion between blocks is obfuscated using our *flatten-and-jump* technique as depicted in Figure 5(b), the static analyzer will report both alias relations <*p, a> and <*p, b> because it does not know which block executes first.

Figure 6 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 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 that controls the flow of execution in the degenerate form of the program 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 identify each subsequent block. While this may allow the adversary to recover some of the original control flow, it is important to note that this analysis requires an interpretation of every preceding block in order to recover the current basic block—an effort that exceeds the cost of most static analyses.

It can also be argued that simulation is required only once for each block, and as a result, the complexity of analyzing such a program lies somewhere between static analysis and a full execution trace, with analysis time proportional to the number of blocks in the program. One way to defeat this analysis is to unroll loops and introduce semantically equivalent basic blocks that will be chosen randomly during execution. Consequently, the effort required in recovering the program control-flow will be comparable to a full simulation. In addition, the initial computation of *swVar* can be erased from memory once it is used to avoid unnecessary exposure of information.

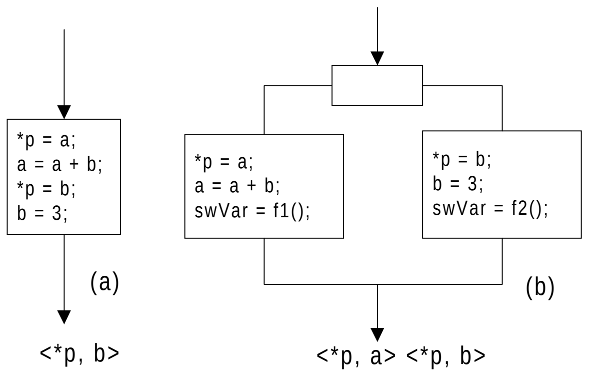


Figure 5: Introducing aliases through pointers

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 determining precise indirect branch addresses statically is a NP-hard problem in the presence of general pointers.

6.1. A NP-hard argument

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-hard in the presence of aliases [12]. Landi later proposed a similar proof to prove that alias detection is NP-hard in the presence of general pointers [16]. The detailed reduction can be found in Appendix A.

The NP-hardness proof establishes that the problem of statically determining branch target addresses is NP hard in the presence of *general pointers*. This result applies to the set of general programs (with general pointers), which, at the first brush of reaction, may not be the same as the set of programs produced by our transformations. We must further establish that the set of transformed programs does not represent a restricted class of programs and that the proof also applies. We approach this as follows.

Assuming the set of general programs is *A* and the set of programs produced by our transformations is *A'*, to show that *A'* is not a restricted subset of *A* (with respect to the NP hard proof) it suffices to show that

1) There is a polynomial time mapping for every instance *a* in *A* to a functionally equivalent instance in *A'*.

2) If there is a polynomial-time algorithm to resolve indirect branch targets for any instance in *A'*, then this algorithm can be used to resolve indirect branch targets for instances of *A*.

Establishing a polynomial time mapping from instances of *A* to instances of *A'* is straightforward; this mapping consists of exactly the code transformations we described in Section 4 and 5.

Because the transformations introduced in Section 4 and 5 are semantics-preserving transformations, an algorithm that resolves indirect branch targets for an instance in *A'* will by definition resolve indirect branch targets for its functionally equivalent instance in *A*. More intuitively, if all the indirect branching targets for an instance in *A'* are resolved to direct jumps, it is a polynomial-time task to restore the original control-constructs (from the flattened *if-else-goto* constructs) and therefore deduce the branch targets for the original program in *A*.

The reduction from 3-SAT does not make use of any program characteristics other than multiple levels of pointer dereferencing and conditional branches. The transformations described in Section 4 and 5 preserve the presence of conditional branches and arbitrary levels of pointers and pointer dereferencing. From an intuitive standpoint, this suggests that the reduction from 3-SAT also stands for the transformed program.

6.2. Complexity evaluation for approximation analysis methods

While the NP-hard result 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 analysis methods: brute-force search and alias approximations.

Brute-force search method. To determine the execution order of the code blocks that appear in the degenerate form of the program, 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 that will be executed. Clearly, this represents the worst-case time complexity and is extremely inefficient when the value of *n* and *k* are sufficiently large.

Alias-detection approximation methods. The problem of precise alias detection in the presence of general pointers and recursive data structures is undecidable [11, 16]. In practice, however, approximation algorithms are often used [11]. An alias analysis algorithm may analyze aliases intra-procedurally as well as across procedural boundaries.

Intra-procedural alias analysis requires as input the alias set holding at the entry node of the procedure, the alias set propagated back from any procedure called within the

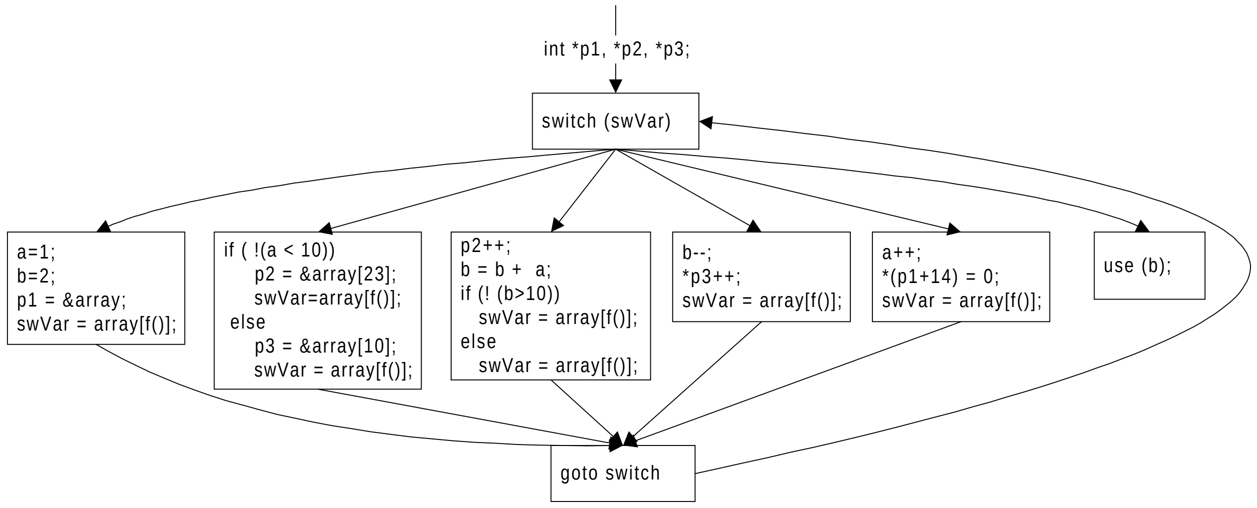


Figure 6: An example transform using pointer manipulation

current procedure, and the alias processing functions (transfer functions) of each pointer assignment statement. Well-known data-flow frameworks [11, 17] exist for handling intra-procedural alias analysis. They are divided into flow-sensitive and flow-insensitive methods. Flow-sensitive methods make use of control-flow path information, and are more precise than flow-insensitive methods. The transformations described in Section 4 and 5 produce a degenerate form of static control flow. As a result, flow-sensitive analysis conducted on this form of control flow loses precision advantage it has over the flow-insensitive methods. Figure 7 illustrates such an example.

The CFG in Figure 7 shows that the assignment *q = &c* overwrites the alias relation <*q, b>, and p = *&*b overwrites <*p, a>. A flow-sensitive alias analysis algorithm, conducted on the control flow in Figure 7(a), would result in an alias set {<*a, c>, <*p, d>, <*q, c>} for this segment. The degenerate control flow in Figure 7(b) essentially represents the set of all possible paths with these blocks. Even a flow-sensitive analysis algorithm at best must conclude with the alias set <*a, c>, <*p, a> <*p, d>, <*q, c>, <*q, b>}. Horwitz [14] presented a definition of *precise* flow-insensitive alias analysis. Under this definition, the flow-sensitive analysis result obtained from the CFG in Figure 7(b) is exactly the same result as a precise flow-insensitive algorithm would have concluded with the CFG in Figure 7(a). We thus conjecture that, with the degenerate form of control flow, a flow-sensitive analysis can be no more precise than its flow-insensitive counterpart.

The transformations presented in this paper are intra-procedural transforms in the sense that they do not affect a control-flow analysis on the procedural level. However, an inter-procedural alias analysis is inherently based on the result of intra-procedural alias analysis, therefore its precision suffer likewise. A step beyond the current scheme is to generalize the transformations to produce degenerate PCGs, which will further degrade analysis results. A detailed discussion on this topic as well as an in-depth study of the complexity of existing alias analysis frameworks can be found in [21].

7. Implementation and Performance Results

We implemented the transformations in a source translator for ANSI *C* in the SUIF programming environment [1]. In our implementation, we developed compiler passes for the code transformations. Each pass traverses the SUIF representation and performs the desired modifications. The exact transformations 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 dead blocks that are added are all determined by a random number generated from the seed.

We tested performance results obtained with experimental transformations on the SPEC95 benchmark programs. Of issue here are three measures: *Run-time performance of the transformed program, performance of static analysis, and precision of static analysis.*

By run-time 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.

7.1. Performance of the transformed program

The following data was obtained by applying our transforms to SPEC95 benchmark programs. Three SPEC pro-

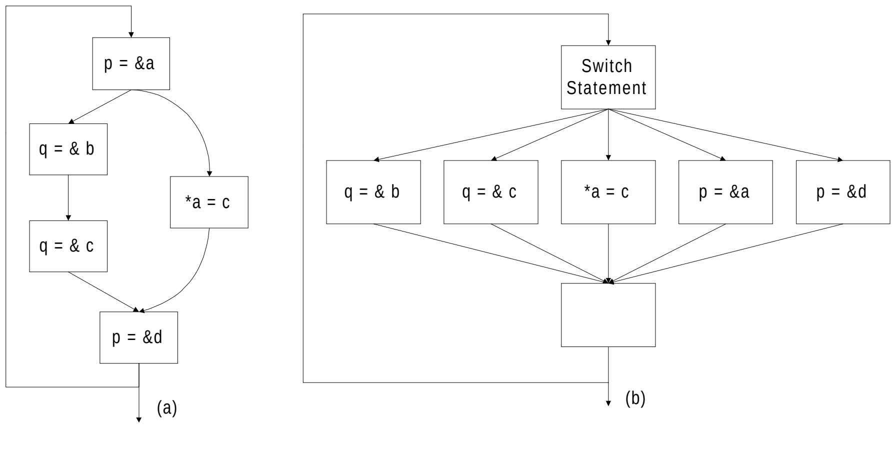


Figure 7: Effect of the degenerate control flow on alias analysis

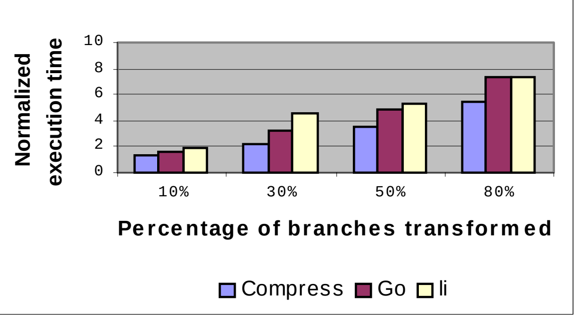


Figure 8: Execution time (non-optimized)

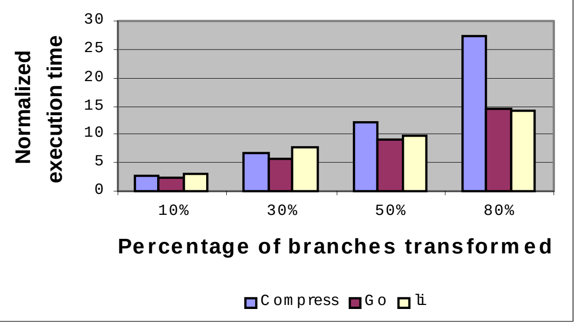


Figure 9: Execution Time (optimized)

grams 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 compression algorithm, and *LI* is a typical input-output bound program for a LISP interpreter. These programs are standard benchmarks used in the compiler community. They embody three major classes of high-level language constructs that are widely used in general programming. It would be more satisfying, however, to test our results on the class of networking programs for which this solution was intended. But in the absence of that, we believe that these test programs 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 speedup due to optimization is significantly reduced when a more substantial portion of the program is obfuscated.

This is an encouraging result; it is highly suggestive (albeit not conclusive) that our transformations 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 the three original programs, compiler optimization performed best on *Compress*—a whopping 80% decrease in the execution time due to optimization. However, as can be seen in Figure 9, our transforms 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 analyses that enabled significant loop or loop kernel optimization were no longer possible after our transform was performed.

The object size of the three benchmarks grew with increased branch replacement (see Figure 10 and Figure 11). *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, and this resulted in less potential for code growth with the transform.

We believe that these results are representative of many programs. It appears that, on average, 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 nearly double in size.

The object size of the three benchmarks grew with increased branch replacement (see Figure 10 and Figure

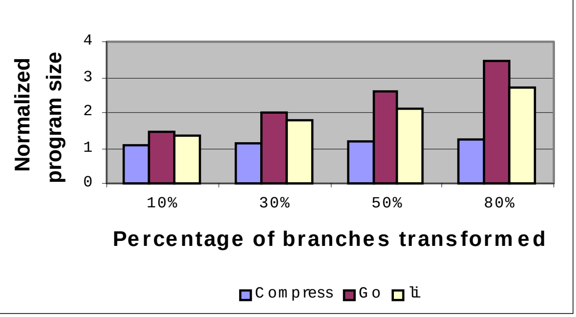


Figure 10: Executable Size (non-optimized)

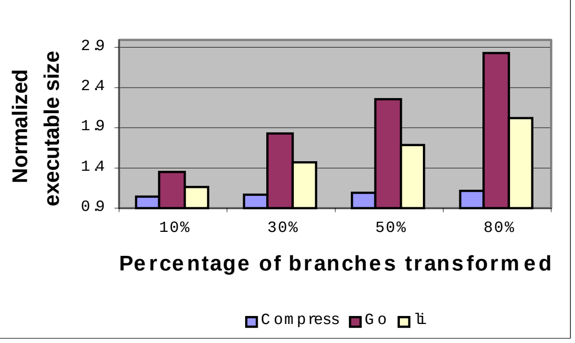


Figure 11: Executable Size (optimized)

11). *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, and this resulted in less potential for code growth with the transform.

We believe that these results are representative of many programs. It appears that, on average, 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 nearly 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.

7.2. Performance and precision of static analysis

In this experiment, we test our techniques against existing analysis tools and algorithms. The state-of-the-art analysis tools include the NPIC tool [13] and the PAF toolkit [18]. They both implement an inter-procedural, flow-sensitive algorithm. Both NPIC and PAF perform control-flow analysis exactly once with no further refinement on the flow graph.

In our experiments, PAF successfully analyzed small sample programs (run to completion) but failed to handle some of the large programs included in the SPEC benchmarks. The failure characteristics were inconclusive as to whether the analysis failed due to difficulties incurred in the alias analysis or an inability of handling the size of the original input program. The test cases that we successfully completed with PAF included a wide range of sample programs that contain extensive looping constructs and branching statements. In each of these test cases, PAF terminated reporting the largest possible number of aliases in the program; in other words, it reported that any pointer variable is possibly aliased to every variable that ever appeared on the left hand side of an assignment statement. 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.

NPIC implements a slightly more aggressive algorithm that includes features such as function-pointer analysis. It performs an iterative analysis interleaving the inter-and intra-procedural analysis. Every time new aliasing information is generated by an intra-procedural phase, it is propagated to its successor functions which then repeat their intra-procedural analysis, and so on, until the alias set converges. Unfortunately, IBM no longer maintains and distributes the tool. The experience with NPIC was therefore limited to analytical experiments with the NPIC algorithm.

A limited number of experiments with the NPIC algorithm were conducted on small programs. These experiments, to the extent that a semi-automated analysis would allow, revealed that little accuracy was achieved when the analysis terminates.

In a particular instance where index computation and aliasing were used to compute branch targets, NPIC started out indicating that the elements of the global array could contain a number of possible values. As the iterations went on, this information was never refined. Rather, alias relations identified in later iterations increased the set of possible values that the array elements were deemed to have. The algorithm eventually terminated and claimed that the elements of the global array were changed an arbitrary number of times, and that they could contain arbitrary values. Computations involving the array elements were deemed unanalyzable. This in turn implied that the indirect branching targets cannot be determined precisely. Alias information propagation among those blocks therefore did not get easier and alias relations were never refined.

8. Conclusion

The problem of protecting trusted software from untrustworthy hosts, is important for many critical functions in modern networks. Consider, as an example, distributed intrusion detection systems in which parts of the ID programs need to operate on untrustworthy hosts. Serious consequences will arise if these programs were the targets of malicious attacks and were compromised.

In this paper, we considered one significant class of attacks, namely those based on static analysis of the binary form of the program. We presented a strategy for defeating analysis by tightly coupling the control flow and the data flow of the program. Since data-flow analysis of acceptable precision is dependent on the control-flow information, this approach is capable of expanding analysis time 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 hard.

We have developed a practical instantiation of the transformation in the form of a compiler for ANSI *C*. The compiler makes a number of changes to the program source including: degeneration of the program control flow; the systematic and general creation of aliases; and the introduction of data-dependent branches. We note that these transformations are not dependent on a *C*-like pointer paradigm—they can be applied to any intermediate representation where explicit memory references exist.

In proof-of-concept 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.

We note that the described transformations produce programs with a considerable level of code diversity (transforms are randomly chosen on a per compilation basis). Such programs, when deployed at various points in a network, are highly resilient to class attacks since most class attacks exploit common software flaws.

It is important to note that the purpose of this work is to eliminate the possibility that a static analysis can be used to deduce useful information for software tampering or impersonation. In other words, the optimal result is that there should be no efficient way to analyze the program other than an actual execution. We also note that many forms of dynamic program analysis make use of static information [3, 10], and the techniques described in this paper will be helpful in defending against these forms of analyses.

Acknowledgments

The authors want to thank Dr. Jim Cohoon for insightful discussions. This effort was sponsored in part by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0314. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

References

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

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

3 Ball, T. and J. R. Larus. “Optimally Profiling and Tracing Programs”, ACM Transactions on Programming Languages and Systems, Vol 16, No. 4, July 1994, pp1319-1360.

4 Collberg, C., C. Thomborson, and D. Low, “Breaking Abstractions and Unstructuring Data Structures”, IEEE International Conference on Computer Languages, Chicago, May 1998.

5 Collberg, C., C. Thomborson, and D. Low, “A Taxonomy of Obfuscating Transformations”, *Techreport 148*, Department of Computer Science, University of Auckland, July 1997.

6 Forrest, S. and A. Soma, “Building Diverse Computer Systems”, in the 1996 *Proceedings of the Hot Topics of Operating Systems*.

7 Hohl, F., “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.

8 Hitunen, M. and R. D. Schlichting, “Adaptive Distributed and Fault-Tolerant Systems” International Journal of Computer Systems Science and Engineering, vol. 11, No. 5, pp. 125-133, September 1996.

9 Knight, J., K. Sullivan, M. Elder, and C. Wang, “Survivability Architectures: Issues and Approaches” in Proceedings: DARPA Information Survivability Conference and Exposition. IEEE Computer Society Press. pp. 157-171.

10 Larus, J., “Efficient Program Tracing”, *Computer*, Vol 26. No. 5. May 1993. pp52-61.

11 Muchnick, S., “Advanced Compiler Design Implementation”, Morgan Kaufmann, 1997.

12 Myers, E., “A Precise Inter-procedural Data Flow Algorithm”, in the conference record of the Eighth POPL. Williamsburg, VA. January, 1981. pp 219-230.

13 Hind, M., 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.

14 Horwitz, S., “Precise flow-insensitive may-alias analysis is NP-Hard”, ACM Transactions on Programming Languages and Systems, Vol 19. No.1, pp 1-6.

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

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

17 Landi, W. and B. Ryders, “A Safe Approximation Algorithm for Interprocedural Pointer Analysis”, Techreport, Rutgers University, 1991.

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

19 Rosen, B., “Data flow analysis for procedural languages”, Journal of the ACM, Vol. 26, No. 2, pp 322-344.

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

21 Wang, C., “A Security Architecture for Survivability Mechanisms”, Ph.D. Dissertation, October 2000, University of Virginia.

22 Wang, C. Hill. J, Knight J. Davidson, J. “Software Protection in malicous environments”. CS Technical Report. CS-00-12. Department of Computer Science, University of Virginia.

Appendix A

Below presents the proof that statically determining indirect branch target addresses in the presence of general pointers is NP-complete.

Proof: Consider the 3-SAT problem for $\Lambda_{i=1}^{n}(V_{i1} \vee V_{i2} \vee V_{i3})$ where $V_{ij} \in \{V_1, V_2, \ldots V_n\}$ and $V_1, \ldots V_n$ are propositional variables whose values can be either *true* or *false*. The reduction is shown in the code below in Figure 12. 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, $**v_1, **v_2, ... **v_n$, *A[];

L2:     A[*true] = &mem1;

L3:     if (-) {$v_1 = \&true; \overline{v_1} = \& false$}  else {$v_1 = \& false; \overline{v_1} = \&true$}
        if (-) {$v_2 = \&true; \overline{v_2} = \& false$}  else {$v_2 = \& false; \overline{v_2} = \&true$}
        ...
        if (-) {$v_n = \&true; \overline{v_n} = \& false$}  else {$v_n = \& false; \overline{v_n} = \&true$}

L4:     if (-)  $A[**\overline{v_{11}}] = \&mem2$
                else if (-) $A[**\overline{v_{12}}] = \&mem2$
                        else  $A[**\overline{v_{13}}] = \&mem2$
        if (-)  $A[**\overline{v_{21}}] = \&mem2$
                else if (-) $A[**\overline{v_{22}}] = \&mem2$
                        else  $A[**\overline{v_{23}}] = \&mem2$
        . . .
        if (-)  $A[**\overline{v_{n1}}] = \&mem2$
                else if (-) $A[**\overline{v_{n2}}] = \&mem2$
                        else  $A[**\overline{v_{n3}}] = \&mem2$
L5:
```

Figure 12: A code segment showing the reduction from 3SAT to static determination of branch targets

Code segment L1 declares the variables and an array A[]. $V_1, \ldots V_n$ are doubly dereferenced pointer variables. L2 indicates A[*true] points to the address of memory location 1 (*mem1*).

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 memory location 2 (*mem2*) 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 $*\bar{V}_{ij}$ is aliased to *true* for this clause. Because every path from L3 to L4 must go through the following statement

If (-) $A[**\overline{v_{i1}}] = \&mem2$ else if (-) $A[**\overline{v_{i2}}] = \&mem2$ else $A[**\overline{v_{i3}}] = \&mem2$

Therefore, at program point L5, A[*true] must point to the address of memory location 2. The code in Figure 12 shows that 3-SAT is satisfiable if and only if the branch target address contained in A[*true] is the address of *mem1*. This proves that 3-SAT is polynomial reducible to the problem of finding precise branch target addresses.