AN ABSTRACT OF THE THESIS OF

Colin W. Van Dyke for the degree of Doctor of Philosophy in
Electrical and Computer Engineering presented on June 06, 2005.

Title: Advances in Low-Level Software Protection

Abstract approved: ______________________________

Çetin K. Koç

Preventative methods for software reverse engineering have been given greater attention in recent times due to the increase in computational resources and tools available to the public. The inherent security provided by encoding source into machine code (executable form) can no longer be assumed, given the availability of effective automated methods for the extraction of source-level structures and information (*i.e.*, intellectual property). Numerous methods have been proposed targeting the protection against reverse engineering tools and techniques; however, one of the most promising and widely used of these is obfuscation, or the introduction of obscurity into a software program. The process of reverse engineering can be seen as a two-phase inverse of compilation composed of disassembly and decompilation. The level of attention that has been given to preventing disassembly through obfuscation is relatively small when compared to the prevention of decompilation. Very few positive results have been published in this arena, leaving it as a promising medium for research. Novel techniques are presented in this dissertation for the prevention of static disassembly on x86 computing architectures. These new methods illustrate two main approaches by which disassembly can be thwarted. Results given within indicate the first positive technique by which the run-time of disassembly is attacked as well as the strongest protection against static disassembly available in current literature. Additionally, new engineering techniques for the realization of effective protection tools are given as improvements over existing methods, leading to a high potency against disassembly at low cost. This dissertation is the first of its kind to address the binary protection of executables against static disassembly and provides a solid ground for future work in this field.

Advances in Low-Level Software Protection

by

Colin W. Van Dyke

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 06, 2005
Commencement June 2006

ACKNOWLEDGMENT

First I must acknowledge Prof. Çetin Kaya Koç for being both a mentor and a friend throughout this entire process. Additionally, this work could never have been completed without the support of Gokay Saldamli, one of the best friends I have ever had. My parents inspired me with the drive necessary for the long journey to this point, and I am forever grateful. Prof. Michael Scott served as a voice of reason during this past year, and none of this would have been possible without him. Onur Aciicmez assisted with the design of a realizable obfuscation tool presented in Chapter Five. Chris Kruegel assisted in the development of a formal platform for results analysis as presented in Chapter Two and Prof. Giovanni Vigna of the University of California, Santa Barbara as well as Prof. Saumya Debray of the University of Arizona contributed their research tools to assist in my results analysis.

DEDICATION

There are three types of people in this world: those that *make* things happen, those that *watch* things happen and those who *wonder* what is happening. This thesis is dedicated to Raymond Wilfred Van Dyke, who taught me to make things happen.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

Advances in Low-Level Software Protection

1. INTRODUCTION

Preventative methods for software reverse engineering have been given greater attention in recent times due to the increase in computational resources and reverse engineering tools available to the public. The inherent security provided by encoding source code into machine code (executable form) can no longer be assumed, given the availability of effective automated methods for extraction of source-level structures and information. Numerous methods have been proposed targeting the protection against various reverse engineering tools and techniques; however, one of the most promising and heavily researched of these is obfuscation.

The process of reverse engineering a software program can be seen as a two-phase inverse of compilation composed of disassembly and decompilation. The level of attention that has been given to preventing disassembly through obfuscation is relatively small when compared to the prevention of decompilation. Very few positive results have been published in this arena, leaving it as a promising medium for research contribution.

In this dissertation novel techniques for the prevention of static disassembly on x86 computing architectures are presented. These techniques illustrate both new contributions to current research in this area as well as the redesign and optimization of existing methodologies. Results given within indicate the first positive technique by which the run-time of disassembly is attacked as well as the best current solution for optimal protection at minimal cost for the reduction of instruction leakage. Additionally, new engineering techniques for the realization of effective protection tools are given as improvements over existing methods.

This dissertation is the first of its kind to address the binary protection of executables against static disassembly and provides a solid ground for future work in this field.

1.1. Summary of Research Contributions

The contributions of this work to the research area of disassembly prevention and low-level software protection are as follows:

- It is the first to develop a method for obfuscation at the assembly-code level targeting binary disassembly, alleviating the inaccuracies of previous techniques in engineering design as well as providing the generation of low associated overhead.
- It is the first to develop obfuscation methods targeting the run-time performance of disassembly.
- A modified branch function design is introduced to be applicable to a broader class of branch instructions within the x86 architecture at lower cost, thus making more rigorous obfuscation feasible.
- It is the first to define a formal, viable method for the analysis of disassembly protection techniques.
- A new, optimized general algorithm for the calculation of junk byte sizes is given.

- It is the first to provide a feasible and effective method for a theoretically *complete* protection of an original source program against disassembly and thus gives the most optimal results to date regarding minimization of disassembly accuracy.
- It is the first to identify the need and ability to perform selective protection of special code regions (algorithms) against static disassembly and provide an architecture for the realization of said techniques.
- It is the first to introduce the problem of *undesirable* code regions as well as identify solutions to this new problem.
- It is the first to give an equation for the computation of Potency vs. Cost for disassembly protection methods, which can be used as a standard metric for evaluating any binary protection scheme targeting static disassembly.

1.2. Background and Motivation

Before reading this dissertation, a certain level of background knowledge is required to understand the contributions developed in subsequent chapters. In this section various concepts are presented in order to familiarize a reader to a level sufficient for understanding later topics at a basic level.

1.2.1. Reverse Engineering

Reverse Engineering, as its name implies, is the inverse of the engineering process. That is, given an end product a Reverse Engineer attempts to evaluate its functionality and determine how its internals are structured in order to obtain some information about the product's design. In this section I will give a fundamental introduction to the reverse engineering process and its application, with specific focus on topics of interest to my research.

1.2.1.1. Applications and Purpose

The purpose of reverse engineering is to extract information about a software program that is not revealed by the developer through any public interface. This information can be of various use to a third-party, and is often gathered for malicious or illegal activity.

Several popular applications of the reverse engineering process include the identification of computer viruses [50, 51], removal of software digital rights management (DRM) [45] and the revelation of exploitable holes in software products [34, 54]. There is no defined application of this concept, and new uses are found as time passes. For example, research in binary code rewriting and optimization [40, 46, 48, 36, 13, 10] is based on fundamental concepts of reverse engineering. However, the roots of modern reverse engineering are seeded in the illegal extraction of information from proprietary software systems. As a result, its legality is often questioned, and new legislation is frequently proposed [22, 4] to combat intellectual property theft from software.

1.2.1.2. Dynamic vs. Static Analysis

There are two fundamental analysis techniques used in the reverse engineering of software. The first, *Dynamic Analysis*, involves the dynamic reconstruction of software internals through direct program execution. Often this is accomplished through the monitoring of current code segments and data values through a debugger.

Static Analysis is the reconstruction of software internals without direct program execution. Rather than observable program behavior, static analysis attempts to reconstruct software internals through the direct evaluation of the binary code. For a more in-depth treatment of these concepts the reader is referred to [33] which provides an illustrated tutorial on both dynamic and static analysis techniques.

The primary application of dynamic analysis is in the isolation of discrete code segments that perform some operation such as serial number registration; in general it is not used for the reconstruction of software internals. Static analysis has a much broader impact in the field of software protection as it builds upon theoretical elements of computer science, such as operating system concepts and programming language design, rather than human interpretation and observation. As a result, software protection techniques targeting specifically static analysis have a stronger theoretical basis and are the target of most current research in this field.

1.2.1.3. Stages of Software Reverse Engineering

The process of software reverse engineering is closely related to compilation [1], in which a series of discrete steps are performed, each producing a lower-level code representation. As stated in [38], reverse engineering can be generalized as the dual process of recovering higher-level structure and semantics from the executable

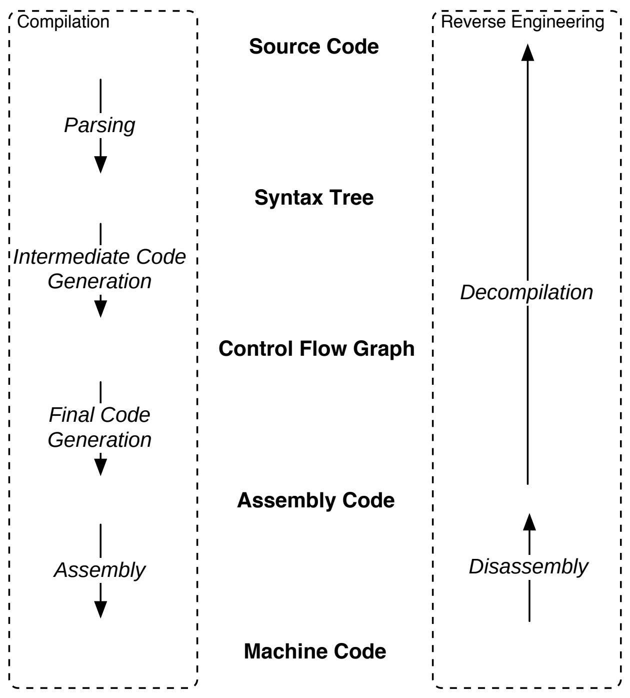


FIGURE 1.1. Compilation vs. Reverse Engineering

machine-code version of the original program. This relationship is illustrated in Figure 1.1 [38].

It can be readily observed that software reverse engineering is broken into two discrete steps: Decompilation and Disassembly. *Decompilation* is the process of reconstructing high-level programming language constructs from assembly language. In general, this is accomplished through a series of analyses that abstract away from the hardware-based features of assembly languages and recover common high-level language features [12]:

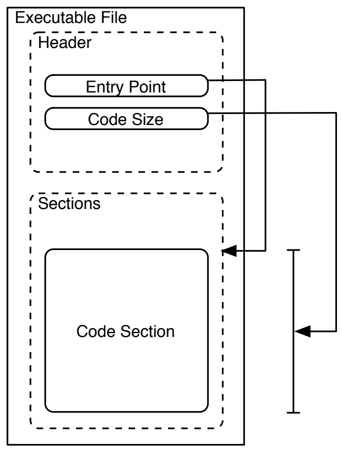
- *Data Flow Analysis*: Recovers high-level language expressions and statements (other than control transfer statements), actual parameters and function return values. Also used to remove hardware-dependant code from assembly, such as register, pipeline and stack references.

- *Control Flow Analysis*: Recovers control flow structure information, such as loops and conditional statements, as well as their nesting level.
- *Type Analysis*: Recovers high-level type information for variables, formal and actual parameter types and function return types.

This area has been researched extensively in recent years [8, 6, 9, 7, 12, 41, 14, 42]. The process of recovering the assembly language used for decompilation from the code region of a binary executable is called *Disassembly.* It is a simple concept complicated by some design constraints in modern computer architecture, and is the primary focus of this research; as a result, it will be covered in greater detail in the next section.

1.2.1.4. Disassembly

Static Disassembly is the process of recovering assembly language instructions from a software program without the invocation of the executable file. Another approach to disassembly is through Dynamic Disassembly, in which a program is executed on some input and each executed instruction is monitored and decoded into its assembly equivalent. Rather than analyzing small windows of code, static disassembly processes an input file as a whole and takes time proportional to the size of an executable program. This is an advantage over dynamic disassembly, which requires time proportional to the number of instructions executed during a program's execution cycle [38].



..

FIGURE 1.2. Executable File Structure

Every executable contains two fundamental parts: the header and sections. Prior to disassembly the actual instruction stream must be recovered from within an executable file, which is accomplished through the location of the code entry point from within the header. This value, coupled with the code size (also indicated within the file header) allows for the isolation an instruction segment from within an executable file. This is illustrated in Figure 1.2 [38].

The disassembly process is complicated by the presence of non-executable data such as jump tables and alignment bytes within an instruction stream, which often causes the production of incorrect assembly instructions [47]. This problem derives from the representation of both instructions and data within a von Neumann architecture in which they are indistinguishable [9]. This is often likened to a chicken-and-egg problem; we are unable to identify instructions without knowing what is data and vice-versa. The problem is exacerbated by statically linked libraries and variable-length instructions. The former forces disassembly to not assume that the code follows familiar source-level conventions or uses known compiler idioms, while the latter complicates the heuristics for instruction sequence extraction. Modern research in disassembly attempts to alleviate these restrictions through direct analysis of machine-code control flow and complex state machines for instruction decoding [47].

There are several published algorithms for static disassembly, formally described in [47], [35] and [38]. These algorithms are the target of the research described in this dissertation and will be presented along with the relevant attacks in subsequent chapters.

1.2.2. Software Protection

The group of mechanisms applied to software programs to aid them against malicious reverse engineering is classified as *Software Protections.* This is a relatively new field of research, whose foundations are in classic virus writing [39]. Most of the techniques used to protect computer viruses in the late 1980s and early 1990s against detection and removal have been applied to the protection of modern software systems against reverse engineering, which share the mutual problem of execution in untrusted environments.

1.2.2.1. General Overview

Proprietary software is distributed in machine-code form, which is fundamentally a low-level representation of the original program. As a result, all potential intellectual property within a software program is distributed in encoded form for execution in what must be assumed as an untrusted environment. To protect this intellectual property, various techniques have been created for making the

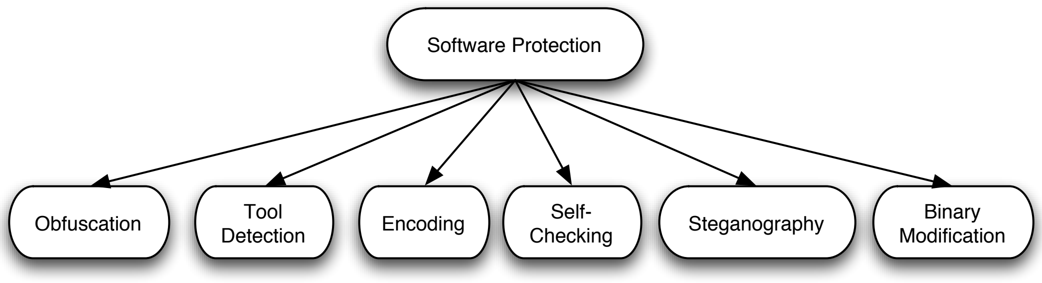


FIGURE 1.3. Software Protection Techniques

extraction of information from binaries as difficult as possible. These techniques, shown in Figure 1.3, have little theoretical foundation and are generally mutually exclusive.

Each protection technique is based on different fundamental paradigms and concepts, and therefore there is no theoretical standard unifying software protection as a field of research. *Obfuscation* is the application of code transforms in order to introduce obscurity into a software program while maintaining observably identical run-time behavior [59, 21, 18, 19, 57, 58, 44]. *Tool Detection* is the process of inserting specific code segments into a software program which use operating system characteristics for the run-time identification of reverse engineering tools such as debuggers [5, 33]. *Encoding* (or Packing) is the application of known cryptographic and non-cryptographic transforms to a software program such that it is distributed in non-traditional (i.e., non-machine-code) form and decoded prior to execution and is a well-known technique for executable polymorphism [33, 39]. *Self-Checking* is the insertion of integrity verification statements within given source code segments that check to verify no modification of instructions or data has occurred [2, 37, 30, 25]. *Steganography* utilizes redundancy in instruction representation and ordering to encode messages within a software program for version tracking purposes (such as identifying the original owner of a stolen software package) and includes new techniques in software watermarking [24, 18, 17, 43]. *Binary Modification* is the most basic protection technique in which physical information is stripped or added to an executable file in order to confuse the reverse engineering process [29].

The most widely used and heavily researched protection mechanism in the above group is obfuscation. Due to its more developed and recognized theory, it will be used as the foundation for our work in software protection.

1.2.2.2. Obfuscation

As previously stated, the most heavily researched software protection technique is obfuscation, whose basic premise is to introduce obscurity into a software program while maintaining identical *observable* program behavior. While not provably secure, obfuscation is seen as the best possible method of hiding information within software without using cryptographic hardware [19].

The concept of program obfuscation spawned from the optimizations applied to a software program by a compiler. In essence, a transform may be seen as the inverse of an optimization for run-time performance. A program is not necessarily streamlined, but a programs obscurity is optimized such that its analysis becomes increasingly difficult. Therefore, some concepts from code optimization can be applied to code obfuscation, such as function inlining and data reordering [19]. The following is a formal definition of an *Obfuscating Transform* used by

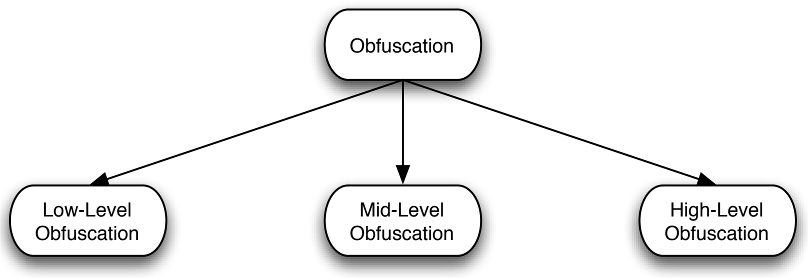
Collberg *et al.* [20]:

Definition 1: Obfuscating Transform
Let $\tau : P \rightarrow P'$ be a transformation of a source program *P* into a target program *P'*. $\tau : P \rightarrow P'$ is an obfuscating transformation if *P* and *P'* have the same observable behavior. More precisely, in order for $\tau : P \rightarrow P'$ to be a legal transformation the following must hold:

- If *P* fails to terminate or terminates with an error, then *P'* may or may not terminate.
- Otherwise, *P'* must terminate and produce the same output as *P*.

A constraint in this definition is the term *observable behavior.* For all purposes of our work, this observable behavior can be defined as the *behavior experienced by a user.* Using this definition, a transform creating a program *P'* may introduce certain non-observable characteristics (such as File I/O, network communication, etc.) that are not contained in *P* and still be considered a valid Obfuscating Transform. In addition, *P'* is not constrained to be as computationally efficient as *P*, and a certain measure of overhead will be inherent in the transform of $\tau : P \rightarrow P'$ [19].

There are three primary metrics that are important in the design of an obfuscating transform as defined in [19]. These metrics allow one to sufficiently evaluate the quality of an obfuscating transform on a superficial level (as there is no sound theoretical basis for the strength of a transform to date). In Chapter 4, a first definite method of evaluation based on these concepts is presented. The primary concerns when evaluating an obfuscating transform are: the level of obscurity added to the program P (a measure of *potency*, M^{τ}_{pot}), how difficult it is to detect and circumvent the transform (a measure of *resilience*, M^{τ}_{res}), and the degree of computational overhead required with the application of the transform



..

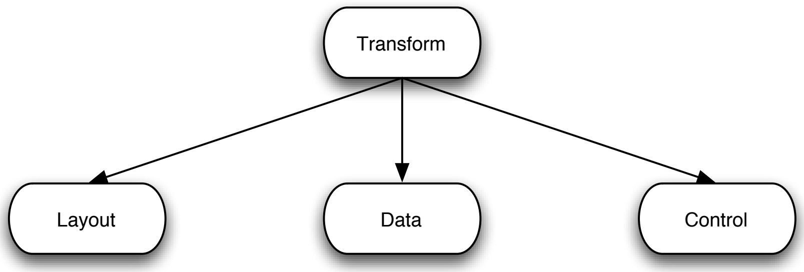
FIGURE 1.4. Obfuscation Type by Realm

(a measure of *cost*, M^{τ}_{cost}) [19].

A great deal of research has been done on developing formal models for obfuscation on different levels. These levels, as illustrated in Figure 1.4, correspond to the different stages that a software program goes through during compilation as illustrated in Figure 1.1. High-Level Obfuscation is the traditional obfuscation of a High-Level Language as illustrated in [21, 18, 19, 57, 58, 44]. Mid-Level Obfuscation targets intermediary languages such as Java Bytecode and Microsoft Intermediary Language (IL) [23]. Low-Level Obfuscation targets machine-code, and has just recently been proven to be feasible in [59].

The primary means of classifying an obfuscating transform is by the type of information it *targets*. This is illustrated in Figure 1.5, and corresponds to the three main classifications developed by Collberg et al. in [19].

The most trivial of obfuscating transforms is the *Layout Transform*, which targets the physical structure (layout) of a source-level program. Some transformations that are classified as effecting layout are identifier renaming and whitespace removal. This is the most widely used technique for source-level obfuscation,



..

FIGURE 1.5. Obfuscation Type by Target

made popular by the annual IOCCC (International Obfuscated C-Code Contest).

Data Transforms target the specific data within a source-level program. Data types carry an intrinsic structure, and this information can be used by a reverse engineer for decompilation. As a result, data transforms attempt to break up this structure into non-traditional representations [19].

Perhaps the most relevant obfuscating transform effects a program's control flow. These *Control Transforms* attempt to obscure the actual flow of control from external monitoring while maintaining an identical run-time behavior. This is an effective method of obscuring internal run-time information (such as algorithms) and is illustrated in detail in [19, 57].

The concept of obfuscation is used as a foundation in the research presented in this dissertation. In subsequent chapters various obfuscating transformations are developed with the specific purpose of targeting a characteristic of disassembly performance. Chapter 2 introduces a new concept in obfuscating transforms targeting the run-time performance of certain disassembly algorithms while Chapter 3 develops transforms introduced in [38] for high reduction of Instruction Leakage at low cost.

1.2.3. Review of Current Literature

This subject of disassembly prevention is a relatively untouched research area, despite its importance to the overall protection of software against attack. Previously there have been only two published advancements [15, 38] which address the problem of thwarting the disassembly of executable code, and they will be identified here. General topics in software protection which form the foundation of these two published results will also be identified.

The first published result was [15], in which Cohen proposed the overlapping of adjacent instructions to fool a disassembler into producing incorrect results. However, no practical implementation of his ideas was provided. Linn and Debray in [38] showed undesirable results in their implementation of Cohen's algorithm. In general, for a successful overlapping of two instructions I and J, they found that the following must hold true [38]:

- Execution cannot fall through from I to J and,
- the trailing k bytes of I must be identical to the leading k bytes of J for some $k > 0$.

These strict requirements were analyzed against a number of software programs, and it was determined that a relative few programs satisfy this criteria in any acceptable manner. For example, Linn and Debray found that in a test of several programs, the program with the highest number of overlaps was *gcc*, which contained only 27 possible overlapping instructions out of a total 360,152 instructions. Cohen's concept of overlapping instructions to fool disassembly, while providing a first positive result in this direction, had very limited impact and therefore was never widely regarded.

The second published result was by Linn and Debray in [38] in which obfuscation was specifically used to target the incorrect static disassembly of executable programs. The primary contribution of this paper was a first valid result in which modern disassembly algorithms are broken by obfuscation. Additionally, self-correction within disassembly on x86 was first identified and briefly analyzed. Several obfuscating transforms are introduced, such as Branch Functions, Jump Table Spoofing and Call Conversion. Performance metrics for the analysis of disassembly-thwarting transforms are introduced and used for the analysis of the previously mentioned obfuscating transforms. This is the only published work in which obfuscation is specifically used to target disassembly, and the authors seem to have halted research on this topic after its publication.

Obfuscation in general is not a new research area, and a considerable amount of work has been published on this topic in recent years. Some of the most paramount work in this area was accomplished by Collberg, Thomborson and Low, who developed a foundation on which to define all later work in this area. In [19], they defined a standard taxonomy, as well as metrics and definitions by which obfuscating transforms can be classified. In [21], they identified new techniques for the construction of opaque predicates, and in [18] they identified and explored three main focal areas in software protection. Other contributions by Collberg et al. to the field of software protection (primarily via obfuscation) appear in [20] and [17]. A parallel work to Collberg et al. was published in [57], and is often used in conjunction with the techniques and algorithms in [19]. One of the primary problems with the techniques outlined in the above publications is the lack of a theoretically strong foundation on which the relative strengths and weaknesses of obfuscating transforms can be analyzed. A recent contribution in [44] approached addressed this issue by developing an obfuscating transform based on a computationally difficult problem.

The focus of current work in obfuscation has mainly been on the prevention of decompilation of a lower-level language in order to extract higher-level semantic information, while little focus has been paid to the prevention of disassembly or obfuscation of machine code. In fact, only in [59] is any attempt made at the obfuscation of machine code in general. Wroblewski's work showed a theoretical algorithm, as well as implementation, for the obfuscation of machine code and proved the feasibility of such a technique.

Hardware-based mechanisms for software protection have been proposed in [2] and [37]. In [2], Aucsmith proposed a method by which software was maintained in encrypted form, and then decrypted during runtime; in [37], specific architectural mechanisms were identified to aid in this type of protection. These two approaches are plagued by the fact that they are infeasible due to high computational overhead without highly specialized hardware, or require a change in the uniform standard computing architecture in order for proper software execution.

Due to the lack of current research on this topic and the relatively small set of publications in this area (two on disassembly prevention [15, 38] and one on machine-code obfuscation [59]), there is a great deal of promise for development and publication of valid results. As stated in [38]: work in this area is orthogonal to the topics mentioned above. That is, any attempt to make a program harder to correctly disassemble will make the process of decompilation have a higher failure rate; additionally it will "sow uncertainty in an attacker's mind about which portions of a disassembled program have been correctly disassembled and which parts may contain errors" [38].

1.3. Disassembly Attack Model

There are two fronts by which disassembly can be attacked such that its performance is adversely affected: accuracy and resources. In this section a formal attack model is developed which will be used to characterize the obfuscating techniques presented later.

1.3.1. Static vs. Dynamic

The focus of work presented in this dissertation is on static disassembly as introduced in Section 1.2.1.4. The problems of protecting against disassembly is split into two mutually exclusive domains according to whether the target algorithm is static or dynamic. Solutions for each domain only apply to their specific algorithm with the exception of *encoding* which can cross domains but is generally considered weak.

Traditionally the problem of protecting against dynamic disassembly is considered unsolvable, due to the availability of on-chip ports which can be monitored to view the currently executing instruction. If the disassembly tool accesses these ports directly, rather than through the operating system, then there is no non-cryptographic method for protection, and cryptographic methods require hardware support as previously mentioned. Most protection methods targeting dynamic disassembly protect only against disassemblers that utilize operating system calls to access the CPU debugging information. By monitoring the values of various system registers after specific interrupt calls, it can be determined if a debugger/disassembler is active on the host system and the software can take adverse action against it. These solutions are highly specific to the environment in which they are run and generally break software portability.

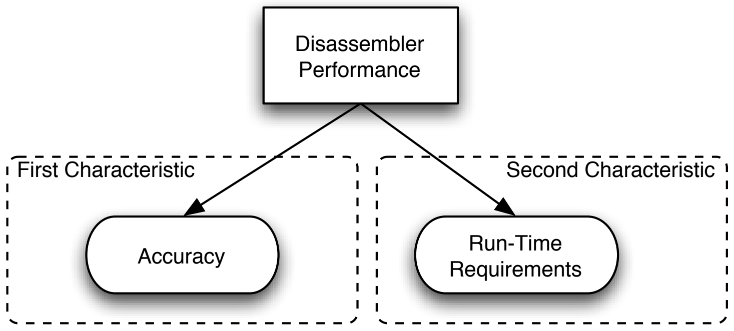


FIGURE 1.6. Static Disassembly Characteristics

Research presented in this dissertation targets static disassembly only as it is not plagued by information leaked during run-time of an executable and is feasibly protected against using non-cryptographic methods. Therefore, the disassembly attack model used considers only static algorithms and does not address any dynamic techniques such that protection techniques are portable across any x86 environment.

1.3.2. Characteristic Targeting

As previously mentioned, there are two fronts by which a static disassembly algorithm can be attacked. To represent this visually, the disassembly characteristics which are targeted by this work are illustrated in Figure 1.6. These two attack domains will be the focus of work presented in this dissertation. To this point there are no published methods other than that given in Chapter 2 that target the first characteristic.

Two attacks are presented in later chapters; the first targets characteristic two, or the run-time requirements of a disassembly algorithm. By forcing the algorithm to require exponential resources based on a binary file size, the use of said algorithm becomes infeasible as time requirements grow. For example, a small file may require a relatively short period of time for disassembly; however, as the file size grows in a linear fashion its requisite disassembly time grows exponentially. The goal of targeting this characteristic is to make the process of static disassembly more expensive than the return on time invested.

The second attack targets characteristic one, or the accuracy of a given disassembly algorithm. This is the traditional goal of any software protection mechanism: to make an algorithm fail to produce correct results. The information presented in Chapter 4 gives an attack against this characteristic. In that chapter obfuscating transforms are presented that severely reduce the accuracy of a disassembly algorithm's results such that use of them would make the reconstruction of higher level structures or extraction of algorithmic information of a binary file inaccurate and incorrect.

1.4. Formal Methods for the Evaluation of Disassembly Prevention

Unfortunately, all current literature in disassembly prevention provides metrics for results analysis but fails to provide a feasible method for obtaining said results. As a consequence, a formal technique for obtaining these results was developed for this dissertation and will be presented here.

To date, the standard and accepted method of evaluation for both disassembly and binary obfuscation tools within the research community has been through the SPECint95 benchmark suite [35, 38, 47]. These benchmark applications are compiled using gcc version egcs-2.91.66 with optimization level -O3. In order to "modernize" the process, the SPECcpu2000 Integer benchmark suite is used as a replacement in this dissertation. Specific compilation or obfuscation steps required by protections given in later chapters will be presented in the relevant sections and omitted here. All simulations are performed on either the Software Attack Machine (SAM), which is a 1.0 GHz Celeron with 512MB RAM running SuSE Linux, or WALES, which is a Dual Xeon 3.06GHz machine with 2GB RAM running RedHat Enterprise 3.

The two common metrics, aside from time/space overhead, used in analyzing binary obfuscation and disassembly are the Confusion Factor (CF) and Disassembler Accuracy (DA), respectively. The Confusion Factor was introduced in [38] and indicates the fraction of program instructions incorrectly disassembled (or, how confused the disassembler was). It is be defined in an abstract form as CF $= \frac{|V-O|}{V}$, where V is the set of valid program instructions and O is the set of disassembled instructions. The notion of Disassembler Accuracy was introduced in [35] and represents the efficacy of a disassembly tool in identifying valid instructions; formally, it is defined as DA = 1 - CF. This metric actually relates the fraction of valid instructions recovered from a binary file.

To measure the effects of of a binary protection technique against various disassembly algorithms, four tools will be used. The GNU Objdump [28] utility is a common development tool implementing the Linear Sweep algorithm. ISLDasm is the ISL Disassembler Suite [32] including tools implementing both Linear Sweep and Recursive Traversal; they were developed strictly for this dissertation (and on a side note, the Recursive Traversal Implementation is the first published for x86/Linux operating on ELF binary files). Objdump and the Linear Sweep version of ISLDasm are not necessarily used in conjunction. Rather, whichever tool achieves the best results will be used for evaluation of protection techniques. The final disassembler was developed by Kruegel *et al.* to illustrate their Heuristic-based disassembly algorithm in [35] and is used only in Chapter 2.

For our purposes a new metric was developed which formalizes the two previously mentioned. This new metric, called *Instruction Leakage* (IL), represents the fraction of original instructions recovered by a disassembler on a protected binary. IL is a scalar value associated with ratio of the number of elements within two defined sets Ω and P. The two sets are composed of unsorted elements ω_j and p_k respectively, where each element is composed of a numeric value (address) and string value (instruction). Instruction Leakage is computed as follows:

Algorithm 1: Instruction Leakage
Formally, define a set element I as $I = \{a, i \mid a \in Z, i \in \textit{x86 instructions}\}$. Then, given two sets Ω, P where $\Omega = \{\omega_j \mid \omega_j \in I\}$ and $P = \{p_k \mid p_k \in I\}$ plus the empty set $S = \emptyset$ we can calculate IL as follows:

1. A new element in Ω is created for each original instruction.
2. A new element in P is created for each disassembled instruction.
3. $S = P \cap \Omega$.
4. IL $= \frac{|S|}{|\Omega|}$

If no element of P is shared with Ω then no original instructions were disassembled (or "leaked"). Otherwise, the scalar value IL will give the fraction of original instructions recovered through disassembly.

Using this abstract algorithm we can derive very specific techniques for accurate analysis of the software protection techniques in this dissertation.

The first applied use of this algorithm is developed for Chapter 2 and is similar to the techniques used in [38, 35]. The premise of this technique is to use the Unix utility *diff*, which gives direct differences between two textual files. From the command-line the steps proceed as follows:

- diff -y <*disassembler output*> <*original instruction reference*> | grep -v '|' | grep -v '<' | grep -v '>' > <*recovered instruction listing*>

This command gives the addresses from <*disassembler output*> that are also contained in <*original instruction reference*>, or the union of the two sets. Instruction Leakage (IL) is then computed through the following pseudocode, which simply gives the ratio of perceived instructions to actual instructions:

```
$count_1$ = wc -l <recovered instruction listing>
$count_2$ = wc -l <original instruction reference>
IL = $\frac{count_1}{count_2}$
```

Output from the disassembler must list only addresses at which instructions are perceived. There is a flag for the PLTO tool [46, 38] which outputs information of the form:

```
feedface -> 0x8000001
0x8000001 -> 0x8000002
```

Instruction mappings which translate an original address to a final address signify mappings of original code and not inserted code. These final addresses are compiled into a list of original instruction addresses used above as <*original instruction reference*> and are used to determine if a disassembled instruction is valid. This technique provides an accurate, applied method for calculating IL with obfuscation based on PLTO [38].

Algorithm 1 is applied secondly to the calculation of IL for topics covered in Chapter 3 by a similar fashion. That is, during the first step of postprocessing for the application of our improved techniques, the original code addresses are generated and later translated according to the location and volume of inserted code/data. The result is an output file that contains original code addresses.

Output of the disassembler is then compared one address at a time to calculate the intersection of disassembler output and original code addresses.

1.5. Problem Statement

The overall goal of this research falls into a single problem: is it possible to make static disassembly of x86 binaries *infeasible*, such that the accuracy of results is minimized or the run-time requirements of obtaining results are exponentially long based on the size of a binary? Additionally, is it feasible to do so such that the overall resource cost of the resulting binary file is minimal? The subsequent chapters in this dissertation address this problem by developing new methods and optimizations based upon obfuscation that give a positive answer to these questions. The results obtained illustrate that both definitions of infeasible within this problem are provided through two different obfuscating attacks.

1.6. Organization of this Document

This dissertation is organized into five main chapters, where each chapter outside of the Conclusion covers a singular topic in low-level software protection.

Chapter 1 presents an introduction as well as the necessary topics forming the basis for the remainder of this dissertation. It begins by defining and developing the topic of Reverse Engineering and then continues to review current techniques in Software Protection. It defines the notion of Static Disassembly as well as Obfuscation, and briefly reviews the current literature on the topic of utilizing Obfuscation for the prevention of Reverse Engineering. A first formal definition of evaluation techniques for binary software protection methods targeting static disassembly is given in Section 1.4. The direct problem statement that this dissertation addresses is given in Section 1.5.

Chapter 2 presents our first approach for the prevention of static disassembly through obfuscation. This chapter introduces a new type of obfuscating transform, classified Structural Obfuscation, and develops a method of thwarting highly accurate static disassembly algorithms that have been robust against current protection techniques [35]. It shows the first published technique by which static disassembly is prevented, not through the reduction of IL, but rather through the increase in disassembly time requirements. It concludes by illustrating the efficacy of protection against disassembly at very small overhead.

Chapter 3 revisits topics presented in [38] and explores methods by which they can be greatly improved. This new design follows the basic concepts in current literature, but recreates the means by which they function in order to increase the efficacy of the concept of Branch Function Obfuscation against static disassembly. Errors and sub-optimal approaches in current literature are identified and modified in order to achieve much more desirable ratios of potency versus cost in the final design. Additionally, new paradigms in the application of obfuscation to the protection of low-level software are given for the first time. This chapter concludes by giving the best current results concerning prevention static disassembly for binary files.

Chapter 4 presents the engineering considerations that are required for the efficient realization of topics covered in Chapters 2 and 3. Fundamental algorithms are developed defining the obfuscation transforms, and tools are presented based on these. Design flaws in previous obfuscation tools are given along with their remedies. Topics presented in this chapter yield a new paradigm in the creation of binary obfuscation tools for the prevention of static disassembly, and give the most minimal resource overhead of any approach in current literature. The chapter concludes by presenting graphically the measure of Potency versus Cost associated with our techniques when compared with others in modern literature on the topic as well as defining a formal equation for the computation of this metric. The equation for the computation of this metric is the first realizable and definite method given in current literature.

Chapter 5 gives concluding remarks on this topic as well as brief review of the results presented in previous chapters. Additionally, future areas of concern are identified and some brief, but not proven, solutions are proposed.

2. STRUCTURAL OBFUSCATION

Most traditional approaches to disassembly followed one of two simple algorithms, Linear Sweep or Recursive Traversal (see Chapter 3). However, with the development of protection mechanisms, which lessened the effectiveness of these algorithms [38], new approaches to disassembly were necessitated. Rather than reinvent the wheel, Kruegel *et al.* proposed in [35] a new disassembly algorithm, similar to existing methods, but designed to be robust and effective for binaries protected with the techniques of Linn and Debray [38]. Their work represented a fundamental shift in disassembly research; rather than focus on good, general algorithms, their approach was to develop a highly specialized algorithm meant to circumvent modern protection mechanisms for binary executables. This keeps with the tradition of software protection in that it mimics an arms race in the Cold War style, by which one side develops a new technology, and the other creates a technique to circumvent it; repeat ad infinitum until a truly unbreakable solution is developed. Their new disassembly algorithm is truly a unique step in a different direction, using code structures and control flow graphs to develop relationships within a program to determine the most likely code sequences for disassembly.

In this chapter, an attack utilizing a new type of obfuscation developed for this research is shown which targets structural assumptions utilized by the algorithm developed in [35]. The process of obscuring control flow boundaries on x86 binaries leads to a decrease in disassembler accuracy and an increase in run-time requirements for this algorithm, both valid methods of weakening the effectiveness of a disassembler as described in 1.3. The results of this work are meant to motivate further research into the robust disassembly of obfuscated and non-obfuscated binary files, as well as break the notion that disassembly time requirements are linear based on input size.

This chapter is structured as follows: Section 2.2 briefly introduces the disassembly algorithm targeted by this work. Section 2.3 presents the topic of Structural Obfuscation and gives an attack targeting pattern-dependent disassembly. Section 2.4 gives the results analysis of this attack.

2.1. Target Algorithm

In [38], Linn and Debray introduced methods of obfuscation specifically targeting the Recursive Traversal and Linear Sweep algorithms through obscuring branch target addresses and inserting non-executable junk data. Their experiments were very positive and provided an effective means of thwarting disassembly. As a result, Kruegel *et al.* in [35] introduced a new, novel disassembly algorithm, based on statistical methods and control graph information, for general and obfuscated binaries. For the remainder of this chapter, their technique will be referred to as *Heuristic-based Disassembly.*

The general method of Heuristic-based disassembly functions much like Recursive Traversal at a high level in that the techniques are based on a program's control flow. However, reconstruction of a control flow graph (CFG) for a given input program is different , and tailored to be more resilient in the presence of obfuscated binaries. From a general view, their approach seeks to isolate the functional blocks of a program through heuristic search for source-level structures indicating a function prologue, and then construct an inter-procedural control flow graph based on the functional subdivisions of the code as well as branch target addresses. This CFG is developed through a multi-pass processing of functional blocks. In general, the first pass develops an initial CFG and further passes attempt to resolve conflicts and ambiguities in the CFG. The remaining portions of a program not covered by the subdivision into functional blocks are processed using statistical methods termed Gap Completion. Additional methods are also used to identify and circumvent the protections developed in [38]. The algorithm developed by Kruegel *et al.* gave very positive results for the static analysis of binaries obfuscated using the methods of Linn and Debray, as well as general binaries. For the purposes of this chapter, a sufficient knowledge of the algorithmic internals is given here. The attack presented below simply obscures actual function boundaries such that the construction of a valid CFG becomes difficult. For a thorough treatment of the algorithm refer to [35], as further development is not necessary in the context of this dissertation.

Static disassembly, in general, is not limited to the brief treatment given here, and several other approaches have been developed based on the fundamental techniques presented above. Schwarz, Debray and Andrews in [47] propose a hybrid method based on the Linear Sweep and Recursive Traversal algorithms. In [11], Cifuentes and Van Emmerik propose a technique for the more accurate extraction of control flow successors for indirect jumps using jump tables; in [10] the same group propose the concept of *Speculative Disassembly*, by which they process undisassembled portions of code in the expectation that it might be a target of indirect jumps and flag it as speculated. The fundamental algorithms of Linear Sweep and Recursive Traversal are given treatment in Chapter 3.

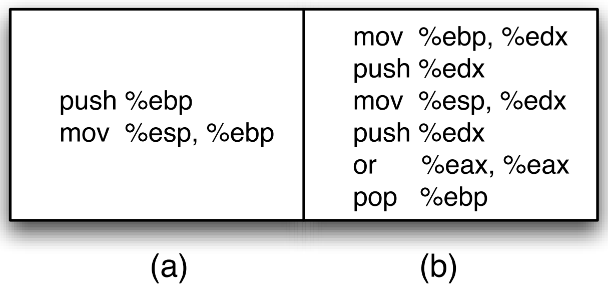
2.2. Utilizing Structural Obfuscation

Several disassembly algorithms used in modern research [35, 47] attempt to divide and process small blocks of code separately within a large piece of software. Primarily, the method used for subdivision of code into smaller discrete blocks is through identification of branch target locations, or pattern matching of known structural source conventions.

All currently published techniques for protection against disassembly exploit the subdivision process used by the recursive traversal disassembly algorithm [38] to reduce information leakage. Structural obfuscation directly targets physical code segments utilized for subdivision and control target identification in certain disassembly algorithms such as [35]. Through our work, we found a desirable attack against disassembly to increase its run-time resource requirements. To date, no such technique has been proposed outside of our research. That is, our work in structural obfuscation is the first approach that gives more weight to making disassembly infeasible due to its execution time requirement, rather than its relative accuracy in decoding. Structural transforms are a special classification of obfuscating transforms wherein code structure is not removed, but translated to its binary functional equivalent as illustrated in Figure 2.1. Additionally, some information may be inserted (given the circumstances) to further confuse function boundaries, similar to the concept of "dead code insertion" introduced in [19]. This is fundamentally different than Layout Transforms defined in [19] in that, while code structure regarding source layout is removed through a Layout Transform (and has no overall effect on the resultant binary), a Structural Transform translates and/or removes structure effecting the physical makeup of a final binary file. Structural Obfuscation is formally defined, using the same concepts of [19], as follows:

Definition 3: Structural Obfuscation Transform
Let $\tau : B \to \overline{B}$ be a transformation of a binary source program B into a binary target program $\overline{B}$. $\tau : B \to \overline{B}$ is a *Structural Obfuscation Transform* if B and $\overline{B}$ have the same observable behavior yet different physical binary structure. More precisely, in order for $\tau : B \to \overline{B}$ to be a legal transformation the following must hold:

- If target structure s_a is present in B and is obfuscated with s_b, then no occurrence of s_a can appear in $\overline{B}$. Additionally,



```
push %ebp
mov  %esp, %ebp
```

```
mov  %ebp, %edx
push %edx
mov  %esp, %edx
push %edx
or   %eax, %eax
pop  %ebp
```

FIGURE 2.1. (a) A simple code structure and (b) its obfuscated equivalent

- If B fails to terminate or terminates with an error, then $\overline{B}$ may or may not terminate.
- Otherwise, $\overline{B}$ must terminate and produce the same output as B.

The notion of using functionally equivalent code sections was first introduced in [24] for the embedding of information within a binary file. Their approach encoded information through various permutations of a short series of assembly instructions. Our methods are much less complicated and essentially an extension of their work to a different arena; we acknowledge that many permutations exist that are functionally equivalent to a select code segment, but do not give weight to one over the other. We randomly select from a pool of functionally equivalent code and replace the original with its functional equivalent to obscure the structure of the final binary program. Given that the subdivision process used by the initial heuristic search of [35] for disassembly assumes certain structural conventions be present within a binary, its run-time performance can be adversely affected through the use of structural obfuscation.

The subdivision technique used by Kruegel *et al.* in [35] is accomplished through identification of function start addresses by heuristic search of a binary file for byte sequences implementing typical function prologues. It is briefly men-

```
    jmp   .fakelabel
    push %ebp
    mov  %esp, %ebp
.fakelabel
```

FIGURE 2.2. A Fake Code Insertion

tioned that confusing the Control Flow Graph (CFG) used for disassembly might adversely degrade disassembler performance. As such, our methods target the confusion of the CFG through obscuring function boundaries at the binary level; that is, we remove the structure of function prologues in the original code and insert "fake" prologues at random intervals within an actual function, essentially complicating the extraction of an accurate CFG for a binary program.

A typical function prologue follows the structure shown in Figure 2.1a, while one possible application of structural obfuscation on this prologue is illustrated in Figure 2.1b; the structure of a "fake" code insertions is given in Figure 2.2. In this work, a set of functionally equivalent code sections of varying size were developed against the patterns utilized for subdivision in heuristic-based disassembly. The structure of a source program is greater diversified with a larger set of equivalent instructions, making the identification of obfuscation techniques scale with difficulty equivalent to the size of the equivalent code set. These techniques were implemented with a tool described in Chapter 4 which parses an assembly file and replaces/inserts obfuscated code structures at given intervals. The impact of removing these structures on heuristic-based disassembly is given in the next section.

TABLE 2.1. Information Leakage (%)

Program	**Original**	**Linn/Debray**	**Structural**
bzip2	97.21	92.68	92.60
crafty	96.17	92.09	92.00
gap	95.66	91.12	89.72
gzip	97.03	92.81	92.61
mcf	97.43	92.91	92.83
parser	96.58	92.50	91.70
vortex	96.17	91.05	89.59
Mean	96.61	92.17	91.58

2.3. Experimental Evaluation

As previously described in Chapter 1, results were obtained for the impact of structural obfuscation on disassembly using the SPEC benchmark suite. These benchmark applications were compiled using gcc version egcs-2.91.66 with optimization level -O3 as done in [35, 38]. These applications are then obfuscated using a structural obfuscation tool, and processed using the methods of Linn and Debray with their tool PLTO [46]. To represent the original version of each tool, the files were compiled as above and not obfuscated ,but simply written into a final executable with obfuscation disabled using the tool in [38]. Each application was tested against the disassembler developed by Kruegel *et al.* in [35]. All results were obtained using an average of ten tests for each benchmark utility on the SAM.

Time and space overhead for our structural obfuscation technique was gathered through a series of ten tests. Overall, the results were not astounding regarding overhead associated with structural obfuscation on the SPECint2000

TABLE 2.2. Disassembler Execution Time (in seconds)

Program	**Original**	**Linn/Debray**	**Structural**
bzip2	50.5	151.8	182.2
crafty	100.9	223.2	2303.33
gap	60.0	181.0	14398.7
gzip	52.8	121.8	206.2
mcf	40.7	94.4	99.7
parser	49.4	136.0	1499.7
vortex	65.0	240.0	18827.3

benchmark tools. The mean execution time required was found to be 2.86% higher than binaries obfuscated with Linn and Debray's approach; additionally, the mean space requirement was found to be 0.40% over Linn/Debray obfuscation. These results indicate a small, statistically insignificant cost associated with structural obfuscation when used in conjunction with Linn and Debray's obfuscation methods.

Table 2.1 gives reference results illustrating the impact of structural obfuscation, in conjunction with Linn/Debray obfuscation, on the accuracy of disassembly. The column associated with **Linn/Debray** gives performance results when a binary has been transformed only by Linn and Debray's techniques, while the column associated with **Structural** indicates performance of binaries obfuscated with the methods of Linn and Debray in conjunction with structural obfuscation. Results for the tool-specific mode of the disassembler are omitted, as this work is addressing the impact against the general algorithm. Kruegel *et al.* in [35] achieved a mean value for information leakage of 90.10 in general mode when operating on SPECint95 binaries obfuscated using Linn and Debray's methods. Analysis for structural obfuscation indicate an accuracy of 92.17 in general mode against the SPECint2000 binaries obfuscated with the same method and 96.61 against unobfuscated binaries. The information leakage achieved against binaries whose structure has been obfuscated is 91.58. W. S. Gossett's (Student's) *t*-Tests were developed to indicate the chance that a standard deviation and calculated mean on small sample sizes (as associated with this research) could deviate from values associated with much larger sample sets (during which true behavior can be seen over a longer time period). The test indicates the statistical significance of variance in small data sets. When using this test on the data in Table 2.1, a t-value of 0.984 was found with a standard deviation of 1.12. We can note that the t-value 0.984 is not greater than the corresponding t-table value (2.18) with probability of 0.05 (95% probability of making a correct statement), indicating that the differences are not statistically significant. This is an important note, as it illustrates an insignificant impact on disassembler accuracy using these techniques and thus they cannot be relied upon for decreasing this metric. However, Table 2.2 illustrates an aspect of disassembly adversely affected by these obfuscating transforms.

A non-traditional method for attacking disassembly is in degrading its run-time requirements. Kruegel *et al.* made a very important point in that their heuristic-based disassembler was not susceptible to the traditional poor scalability and inability to deal with real world input associated with static analysis tools. They indicated that subdivision allows for good scalable performance results, but state that if this is not possible then their disassembler could slow and give no simulation numbers illustrating the degree of impact. One of the goals in developing a valid software protection technique is to make the recovery of code from binary files not only inaccurate, but computationally expensive; the disassembler of Kruegel *et al.* showed a linear increase in time requirement based on the size of the input given accurate subdivision. The structural obfuscation attack, in addi-

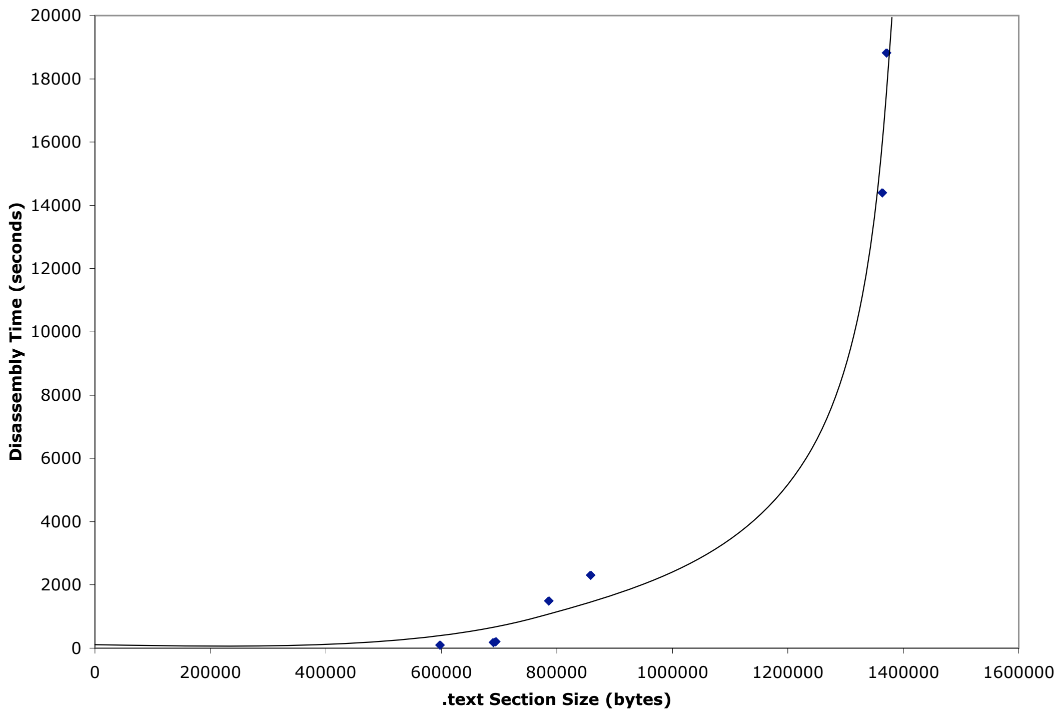


FIGURE 2.3. Disassembly Time Requirements

tion to reducing information leakage, also targeted the run-time requirements of heuristic-based disassembly. The direct impact of this attack on the time requirements of our target disassembler is given in Table 2.2. To represent the numbers visually, the increase in time required for disassembly versus the binary file's text section size is graphed in Figure 2.3. As illustrated, the run-time requirements for disassembly fail to run in a linear fashion, and actually require an average-case exponential increase in time based on the size of the input. This attack effectively makes this disassembly technique infeasible as program size increases. From the results illustrated in this section it is shown that this attack leads to a slight, undesirable (yet statistically insignificant) reduction in information leakage as well as an exponential increase in disassembler runtime proportional to input program size.

The exponential increase in disassembler time requirements is due to the extension of several computationally expensive steps in the disassembly algorithm. The algorithm is cornered on the development of an accurate CFG by which discrete regions of code are developed and determination of correct instruction sequences are based. As we obscure function prologues, they are missed by the subdivision process utilized in CFG generation. Rather, these function heads are interpreted lie within the basic block of a previous function. The insertion of "fake" prologues inside a real function block obscures the boundaries of that function, making it appear as if the original function is not present, but several functions exist within the original function body. Thus, the actual utility calculating the CFG is forced to evaluate erroneous blocks equivalent to the number of inserted prologues. The adverse affect this has on disassembler performance can be evaluated mathematically. Let us assume that the algorithm has to process n blocks of code, and compare their control flow with each other block. The authors of [35] claim this algorithm to run in linear *O(n)* time. The insertion of a single fake prologue will force the processing of additional blocks on the order of $O(n \cdot \frac{n}{2^1})$, as each n functions must be analyzed against $\frac{n}{2}$ new functions to develop intra-procedural control flow. The additional of further fake prologues forces the algorithm to converge to the value $O(\frac{n^m}{2^{m-1}})$. The value m is variable for any given obfuscation attempt, and is in essence a random number. It can be observed that the CFG evaluation is forced to exponential time requirements. This situation is exacerbated by the Gap Completion step, which is the second computationally intensive step of this disassembly algorithm. The level of gaps increases linearly with the value m, which introduces overlapping blocks of ambiguity in control flow and thus several sequences of *candidate* code blocks. The statistical analysis is then forced to determine which block is "most likely" given the pre-computed CFG, and this additional overhead is a constant addition to the above average-time case.

The combination of these two factors force disassembly time requirements into an exponential fashion, and are illustrated analytically through our work.

3. BRANCH FUNCTION OBFUSCATION

The level of attention that has been given to preventing static disassembly through obfuscation is relatively small when compared to the prevention of decompilation, as previously mentioned. The traditional approach has focused on jumping around fake code regions, thus misdirecting a reverse engineer as to what is actual versus "fake" code [5]. Some approaches were proposed in the past exploiting structural aspects of an instruction set architecture [15], but were not widely used due to lack of applicability. The most influential contribution to this area was recently published by Linn and Debray in [38]. The primary result of their work was a first valid method by which modern disassembly algorithms are broken through obfuscation. In addition, self-correction within disassembly on x86 was first identified and briefly analyzed; performance metrics for the analysis of anti-disassembly obfuscating transforms are introduced and used to analyze the methods they proposed.

In this chapter numerous improvements on the fundamental ideas of Linn and Debray are given, allowing for an effective and theoretically complete prevention of static disassembly on x86 binaries. This work builds upon a simple foundation, given in [38], and extends their work through optimization and correction to allow for a greater level of obfuscation to be applied on a binary file, such that the overall accuracy of disassembly is minimized.

This chapter is structured as follows: Section 3.2 introduces the disassembly algorithms targeted by this work, as well as existing obfuscation techniques for disassembly prevention. Section 3.3 presents fundamental improvements and additions to Branch Function Obfuscation and Section 3.4 gives the results analysis of this work.

```
global startAddress, endAddress;

proc LinearSweep(*address)
 begin
  while (address < endAddress) do
   instruction := decode instruction at address;
   *address := *address + instruction.Length;
  od;
 end;

proc main()
 begin
  startAddress := address of first executable byte;
  endAddress := startAddress + sizeof(text section);
  currentAddress := startAddress;
  LinearSweep(&currentAddress);
 end;
```

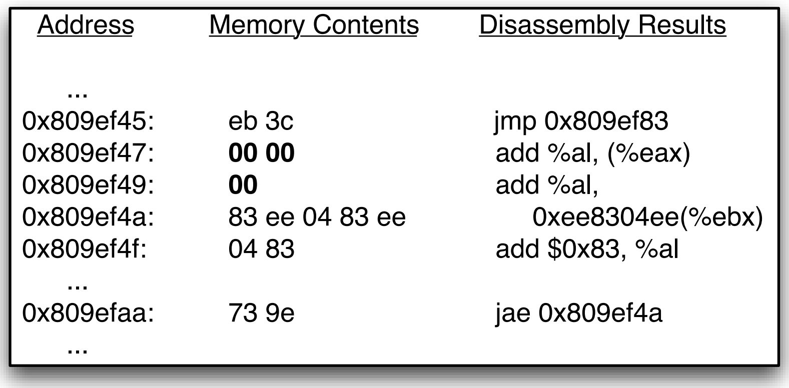
FIGURE 3.1. Pseudocode for Linear Sweep Algorithm

3.1. Target Algorithms

Unlike the previous chapter, the attack presented here targets the two most fundamental algorithms used for disassembly. Consequently, the impact of this work is broader than Structural Obfuscation, as a greater array of commercial tools utilize the two algorithms presented here. Each algorithm has a weakness, which is exploited by Branch Function Obfuscation, and will be illustrated in this section.

3.1.1. Linear Sweep

The Linear Sweep Algorithm starts at the beginning of a code region and sequentially decodes addresses until the maximum address is encountered. This is illustrated through pseudocode in Figure 3.1 [38].



```
Address          Memory Contents       Disassembly Results

  ...
0x809ef45:         eb 3c                    jmp 0x809ef83
0x809ef47:         00 00                    add %al, (%eax)
0x809ef49:         00                       add %al,
0x809ef4a:         83 ee 04 83 ee               0xee8304ee(%ebx)
0x809ef4f:         04 83                    add $0x83, %al
  ...
0x809efaa:         73 9e                    jae 0x809ef4a
  ...
```

FIGURE 3.2. Linear Sweep Example

This algorithm is implemented in several popular disassembly applications such as GNU *objdump* [28], as well as several link-time optimizers such as *alto* [46], OM [49] and Spike [16]. The primary benefit of Linear Sweep is simplicity. However, this algorithm will interpret any data embedded in the instruction stream as a valid instruction. The only exception to this is when an invalid opcode is detected, at which time an error can be flagged. It is due to this reason that it is not considered a "good" technique for disassembly [47].

It is important to look at a simple example of this misinterperetation. Observe Figure 3.2, which provides a simple illustration [47]. At address 0x809ef47, three null alignment bytes were inserted in the instruction stream to push the loop header at 0x809ef4a forward (for reasons such as those outlined in [26]). The Linear Sweep algorithm interprets these bytes as valid instructions, and disassembles as shown. It is easily determined that this is an erroneous disassembly, because the jump at 0x809efaa targets a location in the middle of an instruction. However, Linear Sweep does not identify this error and produces invalid results [47].

3.1.2. Recursive Traversal

The primary problem with Linear Sweep is that it is unable to discern that the alignment bytes in Figure 3.2 are unreachable during execution. Observation of the control flow within the instruction stream would reveal this, and is the technique used by the *Recursive Traversal* algorithm [47].

The Recursive Traversal algorithm is a simple and effective method for avoiding the disassembly of data that is unreachable by program control flow. It is less widely implemented due to its increased complexity, but is seen in several popular binary translation systems [9, 53]. This algorithm functions much like Linear Sweep; however, when a branch instruction is disassembled, the algorithm proceeds to process all possible control flow successors (branch targets). Figure 3.3 shows pseudocode for the Recursive Traversal algorithm [47].

The pseudocode for Recursive Traversal shows that several additional steps are taken for disassembly when compared with Linear Sweep. First, since disassembly is no longer processed in a linear fashion, each address is flagged as being visited if seen by the algorithm. Once decoding on a particular address is finished, it is flagged to ensure that it is not processed further. If this address decodes to a branch instruction, the algorithm recurses to all possible branch targets before continuing disassembly of the next physical address.

It is assumed by this algorithm that the consideration of control flow within a binary will allow disassembly to *go around* data embedded in the instruction stream. The primary drawback of Recursive Traversal results from its most fundamental assumption: that the set of control flow successors for each branch can be accurately identified. This is not necessarily the case, and is difficult in the presence of indirect jumps. This concept is used as the primary means of attack

```
global startAddress, endAddress;

proc RecursiveTraversal(*address)
 begin
  while (address < endAddress) do
   if (address has been visited already) then return fi;
   instruction := decode instruction at address;
   set address as visited;
   if (instruction is a branch or function call) then
    for each (possible target t of instruction) do
     RecursiveTraversal(t);
    od;
   else
    *address := *address + instruction.Length;
   fi;
  od;
 end;

proc main()
 begin
  startAddress := address of first executable byte;
  endAddress := startAddress + sizeof(text section);
  currentAddress := startAddress;
  RecursiveTraversal(&currentAddress);
 end;
```

FIGURE 3.3. Pseudocode for Recursive Traversal Algorithm

against disassembly [38].

Several advancements have been proposed to the algorithms identified here. Schwarz, Debray and Andrews in [47] propose a hybrid method based on the aforementioned algorithms. In [11], Cifuentes and Van Emmerik propose a technique for the more accurate extraction of control flow successors for indirect jumps using jump tables; in [10] the same group propose a new concept of *Speculative Disassembly*, by which they process undisassembled portions of code in the expectation that it might be a target of indirect jumps and flag it as speculated. These algorithms are fairly new and have not been researched extensively [38].

3.2. Branch Function Obfuscation

Linn and Debray published in [38] the first positive application of obfuscation targeting the prevention of disassembly on x86 binaries. Their work, though simple in concept, gave birth to a new arena of research in software protection at the binary level. No longer were protection techniques limited to encryption and encoding, but simple obfuscating transforms and information insertion gave practical results, without necessitating special hardware or software mechanisms for expensive mathematic encoding/decoding. In this section, their work will be introduced such that the improvements outlined later in this chapter become apparent.

3.2.1. Junk Bytes

The accuracy of disassembly is based on alignment of potential instructions with actual instructions. When a sequence of bytes is to be potentially decoded into an instruction, success depends on the first byte of the proposed sequence being the first byte of the actual machine code instruction. As previously stated, this

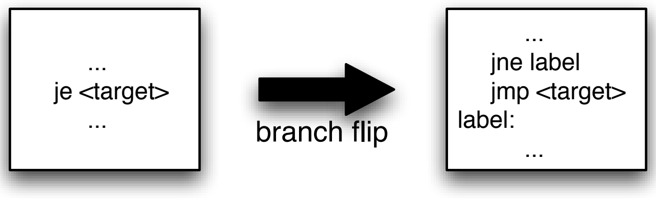


FIGURE 3.4. Branch Flipping

alignment is lost with the Linear Sweep algorithm when some form of unreachable data is encountered. Linn and Debray proposed, in [38], the insertion of *Junk Bytes* into an instruction stream in order to confuse instruction boundaries, and thus produce incorrect disassembly. These Junk Bytes are restricted to satisfy the following criteria [38]:

- (1) They must be partial instructions to confuse the disassembler and,
- (2) must be inserted such that they are unreachable at run-time.

The latter requirement led to the definition of *Candidate Blocks* within a program, after which junk bytes could be inserted without being reachable at runtime. Linn and Debray target blocks of code that begin with unconditional control transfer, such as function calls and direct unconditional jump instructions. They extend the set of candidate blocks by performing *branch flipping* on conditional control transfers; that is, they transform conditional branches to their inverse, and insert an unconditional control transfer to create a new candidate block as illustrated in Figure 3.4. The insertion of junk bytes into candidate blocks is illustrated in Figure 3.5. Junk byte insertion at these locations will break disassembly alignment and produce incorrect results. However, as identified in [38], disassembly will eventually realign and begin producing correct results again. It is therefore necessary to determine the ideal amount of junk bytes to insert at a given code

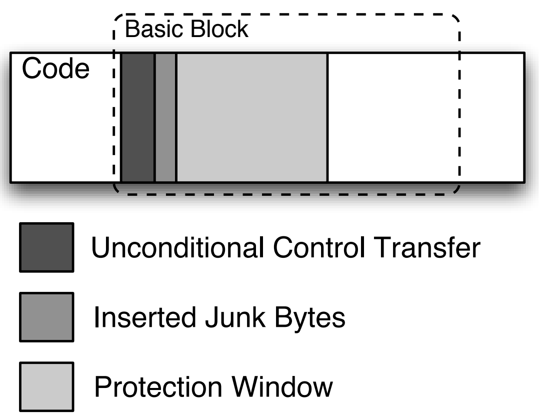


FIGURE 3.5. Structure of Junk Byte Insertion

location to achieve maximally large windows of unaligned disassembly.

There is no formal algorithm for determing that number of junk bytes to insert at an arbitrary control-unreachable location within a program; however, the number of junk bytes inserted should provide the maximum window of unaligned instruction decoding prior to realignment. In [38], a generalized method of determining a value k (the number of junk bytes) is determined as a subset of bytes from an n-byte instruction I (the general instruction XOR was used to supply junk bytes by Linn and Debray). To determine this value, k bytes are inserted prior to a candidate block, and disassembly on that block is simulated. That is, the first k bytes of I are processed by the disassembler, followed by the machine code bytes of the candidate block. Through successive variations of k, and simulations of disassembly on a candidate block of code, the value of k that provides the largest window of broken disassembly is chosen as the ideal number of junk bytes for insertion prior to the given candidate block [38].

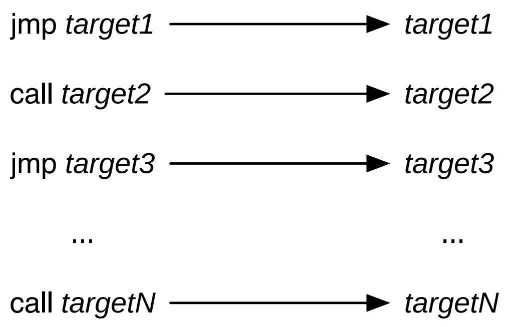


FIGURE 3.6. Structure of Original Control Flow

3.2.2. Branch Functions

The most fundamental assumption used by the Recursive Traversal algorithm is that control flow will continue at the next address following a function call on return. To exploit this the concept of branch functions was proposed in [38]. The fundamental premise of this idea is to convert any unconditional direct-addressed control flow transfer (such as a jmp instruction or function call show in Figure 3.6) to a call to a single *branch function*, which would then redirect control to the original target as shown in Figure 3.7 [38].

Immediately following the branch function call are the junk bytes described in the previous section. The branch function itself rewrites the function return address such that these bytes are "skipped" over, and never executed during runtime control flow. The branch function itself generates control target addresses by indexing a lightweight hash table, whose correct entry is extracted based on a hash of the invocation address. The value obtained from the table is an offset from the invocation address, not a direct target address; the final address is then determined as <*invocation address* + *table offset*> [38].

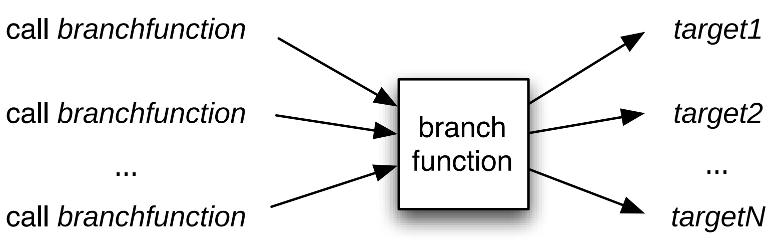


FIGURE 3.7. Structure of Branch Function Control Flow

This technique provides an effective means of exploiting the assumptions of the recursive traversal algorithm. By including junk bytes following branch function calls any linear decoding of machine code instructions is broken for a period before realignment and the control flow of a program is obscured through multiple layers of indirection. The branch function further enhances this security by using a lightweight one-way hash to extract branch target offsets from a table.

3.3. Improving Branch Function Obfuscation

The work of [38], while of paramount importance to software protection, is still immature in development. As a result the work in this chapter is meant to further refine and optimize the basic existing foundation of Branch Function Obfuscation into a strong and effective means for protecting a binary file against disassembly on the x86 architecture. In this section several problems with existing Branch Function Obfuscation methods are presented along with their solutions; additionally, some new techniques based on these concepts are developed. This work furthers the existing knowledge and techniques concerning disassembly prevention through obfuscation and provides the most positive technique for minimal recovery of original code through static disassembly in current literature.

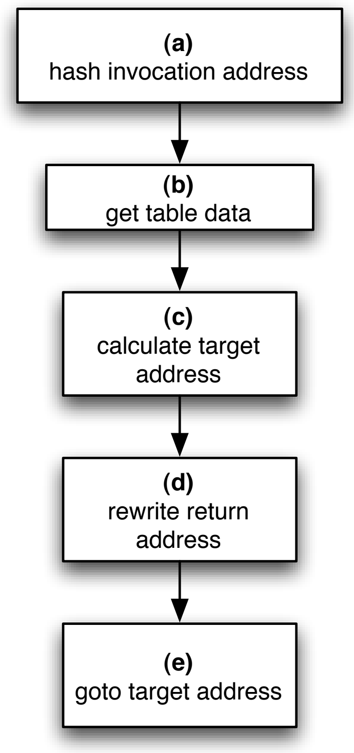


FIGURE 3.8. Traditional Branch Function Design

3.3.1. Branch Function Design

While the Linn/Debray methods of binary obfuscation against disassembly provide very positive results, their techniques provide no means for the protection of windows of code not encompassed by disassembly misalignment. Additionally, they introduce unnecessary complexity as a solution to the attack paradigm of static disassembly. A natural optimization of their work is to reduce the complexity of their branch function design as well as stringent application requirements, such that it supports a broader class of branch instructions for optimal application to real-world binary executables.

The basic design of a branch function used in current literature takes the structure given in Figure 3.8. Function calls and unconditional jumps within a binary file are replaced by calls to this function. The branch function begins by recovering the invocation address, and then hashes it with a lightweight hashing algorithm given in [27]. This value is used to look up a relevant index within a table containing an offset value. The offset is then added to the invocation address, and the branch function redirects control to this point [38].

There are two primary drawbacks when analyzing the branch function design used in [38]. First, it supports a limited subclass of branch instructions; that is, when a conditional branch is encountered, it must be flipped and an unconditional branch is inserted. This leads to information leakage about control behavior if the flipped branch fails to lie within a protection window and the recursive traversal algorithm will disassemble the new target block. The current branch function also has no ability to deal with indirect jumps commonly encountered with the inclusion of dynamically linked libraries.

Second, the inclusion of the hash table and hashing functionality gives unnecessary complexity and run-time overhead to the protected binary. Since omitting protection from being applied to frequently executed regions of code is sub-ideal from a security point of view, it is necessary to limit the computational overhead within a branch function. The purpose of including a hash-table of address offsets is to limit the amount of information available to an adversary. However, this type of attack is not within the scope of static disassembly, which focuses on composed, algorithmic decoding of machine code into assembly language. An attack to extract branch target addresses would require direct binary analysis for numeric patterns, and is not related to disassembly. If the primary attack model concerns static disassembly, then this approach adds unnecessary

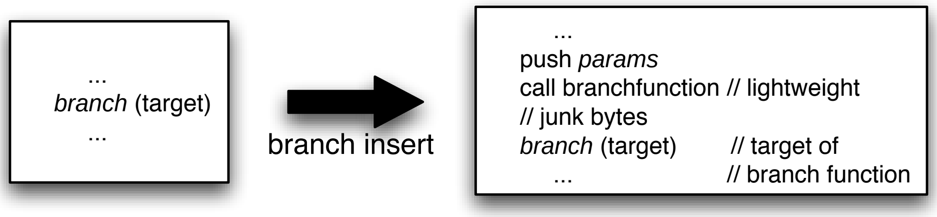


FIGURE 3.9. Lightweight Branch Function

complexity to the branch process and excessive storage requirements on the binary file, while providing no additional security for the targeted attack.

To alleviate these drawbacks. a new lightweight branch function design is proposed; the primary difference is how exactly a branch is protected. Instead of converting branches and cataloguing target addresses through a hash table, this new technique simply ensures that disassembly is always unaligned as it passes control-altering instructions. such that they are never decoded properly. This is accomplished by the *insertion* of a lightweight branch function (as well as junk bytes) immediately prior to control altering instructions. The branch function then transfers control past the junk bytes to the original branch. If this structure is encountered by a target disassembly algorithm. it will be unable to resolve the branch target and move to the next instruction (which will offset disassembly due to the presence of junk bytes). This process, though simple, alleviates many of the drawbacks in previous methods; a branch function insertion is illustrated in Figure 3.9.

The lightweight branch function shown in Figure 3.9 follows an extremely simple design. Its sole functionality is to take a parameter passed to it. representing the inserted junk bytes. and indirectly jump to it (thus making the determination of the branch target infeasible). One possible criticism of this approach is that it exposes a great deal of information passed as parameters to the branch function. However, with the exception of the first branch function call in a code segment. these parameters will be "obscured" by the protection window of a previous branch function insertion. and disassembly will fail to recover them. The only feasible way of recovering these addresses is through a binary file search for patterns indicating a protection insertion, which is out of this problem domain. Regarding the initial branch function insertion used to start disassembly misalignment, a special branch function can be constructed knowing this offset as a static value. Because these branch functions utilize indirect jumps. it is not possible for a static disassembly algorithm to determine the target address. The primary benefit of this approach is in reducing computational overhead associated with traditional branch functions while increasing the class of instructions that can be protected. This approach is not dependent on any conversion, and can thus be used to protect any branch type. When used in conjunction with a new application algorithm given in Section 3.3.4. this method becomes a viable and secure approach for software protection. Additionally, it requires no space overhead when compared to traditional branch functions; that is, those branch functions required a parameter passed to them indicating a junk byte offset as is used in this approach. leading to identical size requirements for calls to the branch function. This new branch function design is more optimal overall for size, as the required code for branch function implementation is reduced.

One goal in this dissertation is to design a branch function that supports any branch instruction and provides a foundation for minimal resource overhead while maintaining maximum protection against static disassembly. The lightweight branch function presented in this section requires minimal computational overhead and, when used in conjunction with the concepts outlined later, allows for aggressive obfuscation such that information leakage can be minimized.

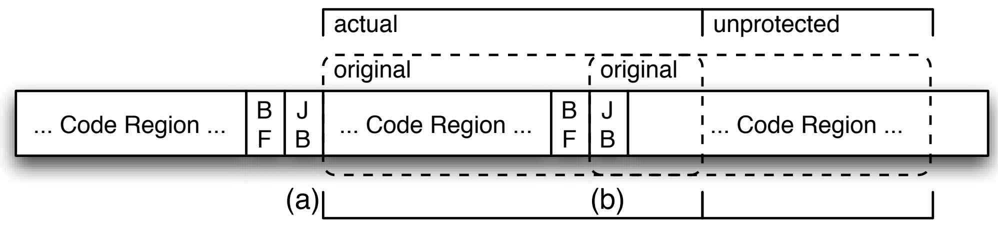


FIGURE 3.10. Original Junk Byte Protection Windows

3.3.2. Junk Byte Calculation

The primary drawback of the approach for calculating the number of junk bytes for insertion following a branch function call provided in [38] is that diminishing returns on size versus protection occur when two or more protection windows overlap, as illustrated in Figure 3.10. Because their tool works in reverse order junk bytes are inserted at (b) such that the protection window generated is maximal according to the aforementioned process. Additionally, junk bytes are inserted at (a) following the same method. However, due to the close proximity of these two code blocks the protection window generated by (a) will overlap that of (b). In this scenario, the protection window generated at (b) is nullified unless this location is the target of another branch traversed by the disassembler. Because the disassembler will *in theory* never process the junk bytes at (b) in their original intention when moving from (a) to (b) this technique of junk byte calculation is limited in this scenario.

Due to this problem, the algorithm provided in [38] is not optimal for size or protection in that there exist situations as shown in Figure 3.10 which do not provide the best calculated size for junk bytes. Through analysis of the instruction profiles of the benchmark tools it was determined that this situation arose with the majority of candidate branches for protection (given the protection characteristics

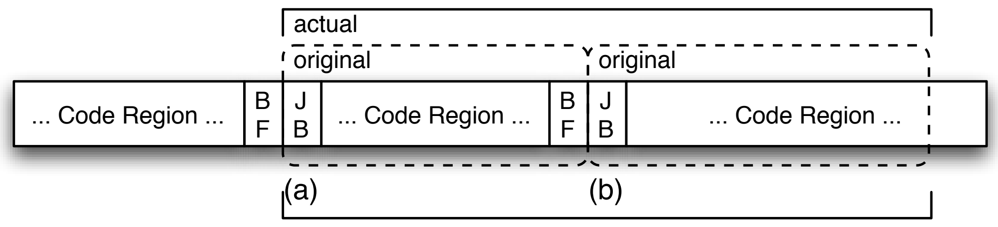


FIGURE 3.11. Optimized Junk Byte Protection Windows

of the junk bytes used in [38]). As a result a simple extension to their algorithm is proposed here to optimize junk byte calculation such that it is more robust. In this new algorithm a value k (the number of junk bytes) is still computed as a subset of bytes from an n-byte instruction I. However, rather than just observing candidate blocks of code a window of m instructions is fetched with the goal of providing the maximum length of protection for m (with m larger than the maximum protection window). The length of the protection window is determined through simulated disassembly as originally proposed in [38]. However, if a branch is encountered within this protection window, the constraints are shortened such that only enough junk bytes are inserted such that protection is provided through the last byte of this branch instruction (where new junk bytes will be inserted for the next protection window). If no branches are encountered then junk byte calculation proceeds as normal to compute the value k providing a maximally large window of protection. Also, if a demarkation instruction (a necessary structure used by supporting research tools for this work presented in Chapter 4) is encountered it is treated as if the immediate ID is zeroed out to ensure consistency with the final executable image. This approach alleviates the problems that can possibly arise in the techniques of [38] and provides a protection characteristic illustrated in Figure 3.11.

3.3.3. Structural Diversification

The algorithm used by Linn and Debray used a single branch function for obscuring branch targets, as well as a single instruction as the source for junk bytes. This approach, while efficient and easily applied, leaks a great deal of information about the protection to a potential adversary. Foremost, it is unusual for a program to redirect all function calls to a singular address. The presence of this behavior within an executable can leak the fact that the file has been protected using known techniques, and the reverse engineering strategy can be modified accordingly. Specifically, an attacker can modify their disassembly approach such as using special algorithms targeting the protection technique [35], or utilizing dynamic disassembly on the exposed branch function to extract branch target addresses.

The use of a single, known instruction for creation of junk bytes is also a weakness of the techniques developed in [38]. That is, the byte sequence used for disalignment of instruction boundaries is predictable. The junk bytes can be identified through heuristic search of the code giving candidate junk sequences whenever the known instruction is incomplete or has improbable parameters, giving information about the obscured branch return addresses.

To alleviate this problem, the structural diversification of inserted information used for disassembly protection is proposed. A multiple number of branch functions should be generated and placed at random address intervals within a binary code section. These branch functions should be functionally equivalent, yet structurally diverse, which can be accomplished using the techniques of [24, 55, 56]. The inclusion of multiple branch targets will diversify the branch target addresses and mimic the structure of an unprotected executable. This approach creates a diverse environment in which no branch “monoculture” exists, hiding as much as possible the presence of protection within a binary file. That is, the branch function calls will appear to be regular function calls, or calls to heavily used functions such as *printf()*, etc.

In addressing the issue of junk bytes, we propose using the entire instruction set architecture as candidate instructions from which junk bytes can be extracted. This has two benefits: first, the identification of junk byte sequences becomes much more difficult and improbable; second, we can search the entire instruction set for the maximal junk byte sequence for any given location, leading to increased protection window lengths and less space overhead overall. Because the cost of any information *insertion* into a binary file is very high in regards to overhead in the final binary, it is important to always look for the highest level of protection while maintaining the lowest cost.

In the development and realization of these methods, a calculation was made of the average maximal window of protection achieved by use of diversified junk bytes for insertion. Contrary to the small windows associated with the single instruction used in [38], our methods were able to achieve average protection windows of much greater length (on the order of approximately 50-90 instructions). This leads to less information insertion, which increases protection cost more than any other facet.

3.3.4. Degree of Obfuscation

The algorithm of [38] applied obfuscating transforms to branch instructions within a binary program. This approach led to a protection profile illustrated in Figure 3.12 due to the limited branches considered valid for transformation. A natural extension of this technique is to insert lightweight branches (where they always are untaken conditional branches) into unprotected code regions such that all orig-

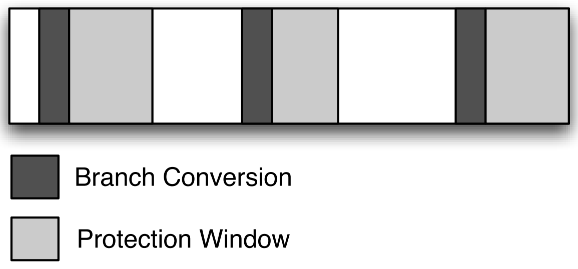


FIGURE 3.12. Original Protection Profile

inal program instructions are protected.

The application algorithm for this is similar to that of Linn and Debray, but if a branch instruction is not encountered at the end of a protection window, a new branch is inserted to break disassembly alignment and create a new window of protected code. This process is repeated until a new candidate block is reached and gives a protection profile illustrated in Figure 3.13. Utilizing this complete protection method, the possibility of leaking parameters passed to a branch functions is nullified; that is, these parameters will always fall within a protection window and their recovery through static disassembly will not happen. The benefit of this approach is that all original code within a binary file can be *theoretically* protected against static disassembly. To formalize this claim, let us generalize the target binary file as a series of basic blocks b_i where the overall executable can be represented as $B = \{b_0, b_1, ..., b_n\}$ and each b_i begins with a branch instruction. Define additionally the windows of unprotected code for the binary file as $W = \{w_0, w_1, ..., w_n\}$ where w_i corresponds to the window of unprotected code in basic block b_i. Assume protections are applied to each basic block b_i and provide protection windows p_i^j where each j states the order the protection was applied in the current basic block. For example, pretend that at basic block b_i two separate protections are applied. They would therefore be denoted as $\{p_i^0, p_i^1\}$.

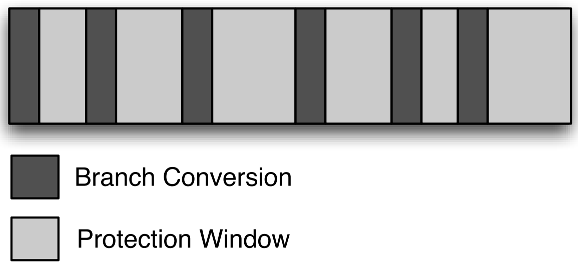


FIGURE 3.13. Modified Protection Profile

Now, assume that a protection is applied at the head of each block b_i providing protection windows p_i^0 of length $\leq (w_i - p_i^0)$. If $w_i - p_i^0 = 0$ then the basic block is fully protected and no windows of unprotected code exist for the current basic block. Otherwise, a window of unprotected code of length $w_i - p_i^0$ exists starting at $b_i + p_i^0$ and must be resolved. In this case, a protection p_i^1 is inserted at $b_i + p_i^0$ and the window of protection is analyzed against the previous test. That is, if $w_i - (p_i^0 + p_i^1) = 0$ then the block is fully protected; if this difference is still > 0 then further protections $p_i^j (j > 1)$ are applied at location $b_i + p_i^0 + ... + p_i^{j-1}$ until $w_i - (p_i^0 + p_i^1 + ... + p_i^j) = 0$. This process is completed for each $b_i \in B$, creating a profile that will theoretically leave no $w_i \in W$ strictly larger value than the corresponding $p_i^0 + p_i^1 + ... + p_i^j$.

One further consideration addressed by this aggressive level of obfuscation is the protection of branch function parameters. In [38] the static offset to the branch fall-through instruction is passed as a parameter to their branch function, effectively giving the location of the next sequential valid instruction. As previously noted, our techniques utilize this approach as well. However, through more aggressive obfuscation it can be ensured that this parameter is never recovered by static disassembly.

To accomplish this, we observe the previously mentioned algorithm for junk byte calculation. Branch function parameters will lie as an instruction immediately before the branch function call. Because protection windows are created such that they protect up to the branch function call instruction, these parameters will be broken from static disassembly. In the situation that a branch function call must be inserted outside of the original branch profile due to insufficient protection windows, the branch function parameters are always included as the last instruction in the protection window, with the branch call occurring immediately after. This profile-independent branch function insertion, however, is rare in practice accounting for less than 5% of actual protections. This can be credited to the use of diversified junk bytes which achieve much more practical windows of protection while applying the same degree of obfuscation provided in [38].

3.4. Experimental Evaluation

As previously described in Chapter 1, results were obtained for the impact of this new branch function obfuscation on disassembly using the SPEC benchmark suite. These benchmark applications were compiled to assembly using gcc version egcs-2.91.66 with optimization level -O3 as done in [35, 38]. These applications are then obfuscated using the Nebbiolo tool presented in Chapter 4 which implements the concepts of this chapter. Each application was tested against ISLDasm, a simple disassembler developed for this research implementing both Linear Sweep and Recursive Traversal as well as the popular utility *Objdump* [28] for results verification, which implements the Linear Sweep algorithm. All results were obtained using an average of ten tests for each benchmark utility on WALES.

Time and space overhead for improved branch function obfuscation was gathered through a series of ten tests. Due to the use of a tool designed to improve these characteristics as well as a lightweight branch function design, the

TABLE 3.1. Percent of Original Instructions Recovered Through Disassembly (IL)

Program	**Linear Sweep**	**Recursive Traversal**
bzip2	02.66	02.89
crafty	05.50	05.74
gap	04.24	04.51
gzip	04.43	04.82
mcf	07.16	07.61
parser	03.04	03.31
vortex	04.10	04.43
Mean	04.45	04.76

resulting size overhead is much less than in [38] and was, in general, 30.87% over the original binary file. Execution overhead was 43.14% on average which provides a beneficial (yet statistically insignificant, as shown in Chapter 4) decrease from [38], due to the use of a lightweight branch function despite rigorous obfuscation. These results are further developed along with the formal tool design in the next chapter.

Table 3.1 gives reference results illustrating the impact of improved branch function obfuscation on the accuracy of the two target algorithms. It can be readily observed that techniques presented here provide a reduction in the accuracy of disassembly protecting original code within a program against both Linear Sweep and Recursive Traversal when compared to [38]. In fact, the only properly disassembled instructions are encountered in areas where junk byte calculation failed to develop an optimal result, leaving open windows of select unprotected code. For Linear Sweep, the average IL value was 4.45 with a median value of 4.24 and a standard deviation of 1.52. Recursive Traversal had an average IL value of 4.76 with a median of 4.51 and standard deviation of 1.57. Previous work in this area [38] has achieved an average information leakage of 15.0% for Linear Sweep and 60% with Recursive Traversal, both incurring an execution penalty of 52% on average. When comparing our results with [38], there is a probability of less than .0001 that the results were achieved by chance and are therefore statistically significant. Given the above results and those presented in the next chapter regarding resource overhead, the techniques given here for branch function obfuscation are a sound improvement in this area given the more aggressive obfuscation; however, the reduction in execution overhead can be considered statistically insignificant as shown in Chapter 4. The reduction in IL for the Recursive Traversal algorithm is due to our application techniques, which ensure that no original branches are "seen" by the disassembler. Additionally, the absence of flipped branches ensures that each window of code is reached from only branch fall-through, which lands directly on inserted junk bytes each time. This leads to profound reduction in accuracy for Recursive Traversal when compared with current literature.

The primary goal in our improvement of branch function obfuscation was to accomplish *complete* protection of a software program against static disassembly. As illustrated above and previously mentioned, this goal was not reached due to *Undesirable* Code. Undesirable code can be defined as a code region in which no combination of available junk bytes was able to achieve a protection window of length 2 or greater. That is, the window head and second instruction were accurately disassembled into a single incorrect instruction prior to realignment. As a result, when this scenario was encountered the instruction was "left behind" and obfuscation proceeded at the next subsequent instruction. The process continued until no undesirable code is found, leaving behind variable-length windows of unprotected code. The solution to this problem lies in extending the class of junk instructions for protection to the maximal count. In our realization, a single example of each instruction is utilized. However, given that instructions can have numerous input values, it was infeasible to implement them all and left as future research. Work in this area must continue to extend the class of valid junk instructions such that they cover all possible instruction combinations (a very, very large number). Only once the entire class of each possible instruction combination is tried can the feasibility of complete code protection be known.

The primary goal in improving branch function obfuscation was to achieve a minimal recovery of original code through static disassembly with accurate and efficient application. From the results indicated in the above table it is seen that improved branch function obfuscation gives a significant improvement over previous work in this area in both strength and cost. Future work in this area should focus on further refining these techniques to achieve full binary protection (as previously mentioned) as well as reduce the overall cost to improve the Potency versus Cost ratio.

4. ENGINEERING DESIGN FOR DISASSEMBLY PREVENTION

One of the most glaring drawbacks of existing binary protection tools was in the actual application of obfuscating transforms; this is not a theoretical problem with their techniques, but rather an engineering design problem. As a result, some of the fundamental contributions of this research to the field of binary software protection is through the development of obfuscation tools and techniques, which are not plagued by certain engineering flaws. In this chapter two tools are presented which were developed to implement the concepts of Chapters 2 and 3 while providing new concepts for the realization of efficient binary protection.

This chapter is structured as follows: Section 4.1 gives the design, algorithm and abstract implementation details for the realization of Structural Obfuscation. The resultant tool, as well as its specific functional algorithm, are the first to develop Structural Obfuscation in current literature. Section 4.2 gives the design, algorithm and abstract implementation details for the realization of an efficient Branch Function Obfuscation tool. This tool is the first to implement the Improved Branch Function concepts of Chapter 3, and gives the best ratio of Potency versus Cost in current literature.

4.1. Tempranillo

Implementation considerations for the obfuscating techniques given in Chapter Three are presented here. This tool, *Tempranillo*, is the first tool designed to perform Structural Obfuscation for the prevention of static disassembly. Its design, fundamental algorithm and abstract implementation are given here while the full source is given in Appendix A.

4.1.1. Design

The Structural Obfuscation tool used as a realization of the protection techniques of Chapter Three was designed to be exploit simplicity and accuracy. Its most fundamental contribution, from an engineering point of view, was that it is an effective obfuscation tool directly targeting x86 assembly, and the first binary protection tool to obfuscate at the assembly level while targeting disassembly execution time. It provides a foundation through which future Structural Obfuscation methods can be refined and realized, and its core libraries serve as the foundation for all tools treated in this chapter.

The only design restriction placed on this tool was that it had to not introduce excessive resource usage into an obfuscated program, and was required to transform the original assembly code to its functional equivalent. This is accomplished this through a static search of the assembly code for known target structures and replacing them with an obfuscated version. Additionally, at random intervals within a function it would insert "fake" code that would make it appear as if a single function was composed of multiple functions without termination (meaning that those functions never returned, just blended from one to another).

4.1.2. Algorithm

Tempranillo follows a straightforward algorithm for the application of structural obfuscation to a set of assembly files that generate a target program. In this section, a theoretical description of the structural obfuscation process is presented.

The *Structural Obfuscation* algorithm is a technique for obfuscating low-level software against malicious disassembly and control-flow analysis through the transformation of its basic structures used by the techniques of [35], and is defined as follows:

Algorithm 2: Structural Obfuscation
The algorithm is given as input F defining a fake code structure, $I = \{i_0, i_1, ..., i_{n-1}\}$ defining a set of n assembly language instructions, x ($\geq n$), $T = \{T_1, ..., T_s\}$ defining a target code structure of s lines and $\tau = \{t_1, t_2, ..., t_m\}$ defining m code structures which is functionally equivalent to T. It then proceeds through the following pseudocode:

```
// (A)
j = 1;
a = j;
b = 1;
r = random(1..x);

// (B)
while (j <= n) {
    // (C)
    if (r mod j == 0) {
        insert fake structure F prior to $i_j$;
    } else if ($i_j$ == $T_b$) { // (D)
        a += 1;
        b += 1;
        matched = TRUE;
        while ((b <= s) AND (a <= n)) { // (E)
            if ($i_a \neq T_b$) {
                matched = FALSE;
                break;
            }
            a += 1;
            b += 1;
        }
        if ((matched == TRUE) AND (b == (s+1))) { // (F)
            c = random(1..m);
            insert $t_c$ at $i_j$;
            j = a;
        }
    }
    j +=1;
    a = j;
    b = 1;
}
```

At location (A), variables are initialized. At (B), the algorithm loops until n lines of code have been visited. At (C), a fake structure is inserted at random intervals. At (D), the current instruction is checked against the first line in a target structure. The algorithm loops at (E) to verify the presence of a target structure. If present, the functionally equivalent is inserted in place of the target structure at (F).

It can be readily observed from the above algorithm that the outside loop executes at most n times. During each path through the outer loop, if the current instruction is equal to the first line of the target structure, then an inner loop is executed at most $s-1$ times. In terms of run-time complexity, the worst case happens when each instruction forces the execution of the inner loop. The outer loop provides a minimum complexity of $O(n)$, while the inner loop adds s iterations in the worst case for each n. The resultant worst-case complexity is then $\Theta(n \cdot s)$. However, this value s is a constant giving an average run-time complexity for this algorithm of $O(n)$. It is the first formal algorithm for Structural Obfuscation as that concept is introduced in this dissertation and in [55] for the first time. The tool Tempranillo implements a version of this algorithm as described in the next section.

4.1.3. Implementation

An abstract implementation overview of Tempranillo is shown in Figure 4.1. It is readily discerned that this implementation follows very closely with the algorithm given in the previous sub-section with the exception of a few architectural considerations. As presented in Chapter 2, this tool provides astonishing results when attacking specific disassembly algorithms.

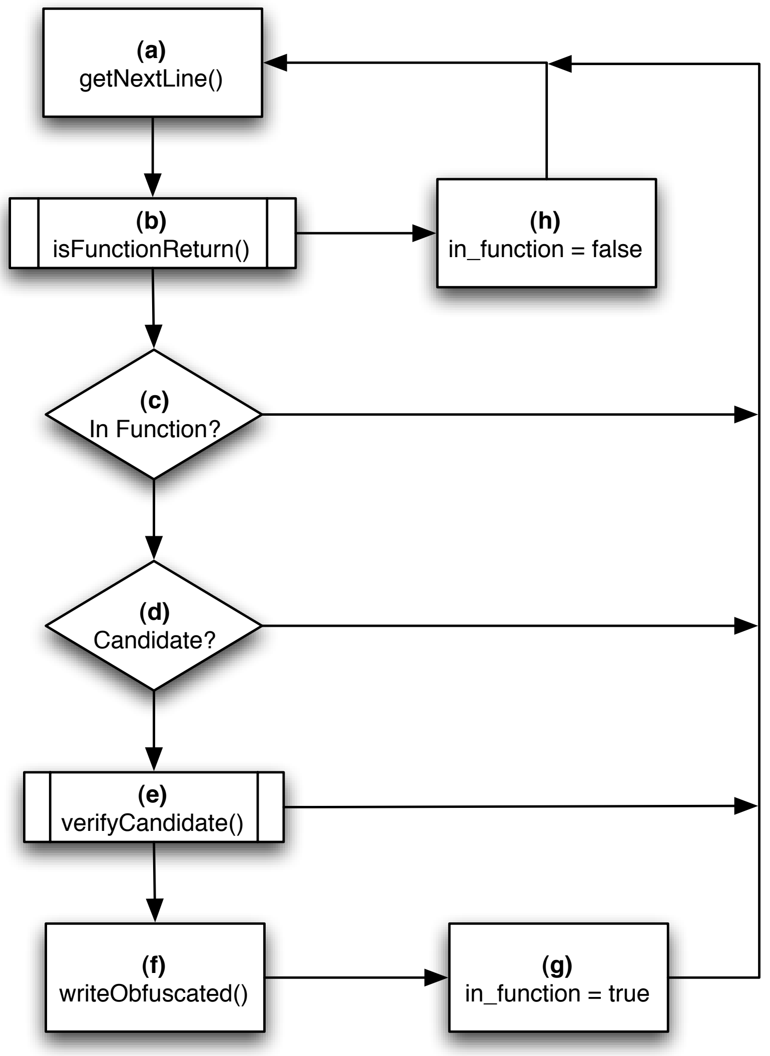


FIGURE 4.1. Abstract Implementation for Tempranillo

The structural obfuscation algorithm is realized through the steps presented in Figure 4.1, which are applied by a single pass through each assembly file within a given program. For each assembly file, the flow of control for structural obfuscation proceeds as follows: at each line within a source file a custom x86 assembly parser performs the function *getNextLine*() at **(a)** which returns a single line of assembly code. At **(b)** the function *isFunctionReturn*() checks to see if the current instruction indicates an exit from a function frame. If true, then the boolean value *in_function* is set to false at **(h)** and control resets. Some transformations for Structural Obfuscation require that they be applied within a function frame; at **(c)**, a check is performed against the value *in_function* which, if true, proceeds to **(d)**. The reasoning for this is that fake code insertions can only take place within a function.[1] If an instruction is equivalent to the first line in the predefined structural profile as given in the formal algorithm (and defined within the file *structures.h*) then it is classified as a candidate at **(d)**. Given the presence of a valid candidate, it must be verified to conform to the specified structure as performed at **(e)**. This verification process simply checks the corresponding lines speculatively following the current source line to conform against the pre-defined structure. If they are verified then the program writes an obfuscated equivalent in their place at **(f)**; if not, then the algorithm repeats. The specific nature of the attack outlined in Chapter 3 dictates that the target structure is a function prologue. If control passes the candidate verification process without error than it is noted at **(g)** that a function has been entered, leaving the possibility for further

[1]Not noted in the ASM, a check of the instruction index (the instruction number relative within the current file) is made between **(c)** and **(d)** against a randomly generated value. If the two are equivalent, a fake structure is inserted to obscure function boundaries as described in Chapter Three.

obfuscation as previously described (through "dead" code insertion).

This implementation allows for a simple yet effective means of obscuring function frame boundaries within a given source program. As described in Section 2, it provides minimal time an space overhead, while giving maximal attack results, against a disassembly characteristic previously untouched (that is, no existing attack methods address this aspect of disassembly). This tool provides the first steps in a new paradigm of software protection.

4.2. Nebbiolo

In this section, the design of an efficient and effective obfuscation tool is developed targeting the reduction of information leakage through new, lightweight techniques in Branch Function Obfuscation given in Chapter 3. This tool, *Nebbiolo*, is a unique approach to creating a practical method of protecting software against static disassembly. The actual realization process changes the fundamental way in which Branch Function Obfuscation is applied to create an ultimate solution giving the most optimal Potency versus Cost to date, and serves as a foundation for future work in this area.

4.2.1. Design

Numerous design constraints for this tool were developed to optimize the application process and introduce minimal overhead into the final protected binary file. Each is a fundamental contribution to this research area making the application of the obfuscating transform as secure and optimal as the protection technique itself.

4.2.1.1. Application Phase

A fundamental change in this design from others is the abstraction of the obfuscation application level. Previous work in this area [38] has operated on binaries specially compiled to contain relocation information for disassembly. This led to unnecessarily large executable size for the level of protection provided; additionally, the approach of [38] did not integrate well into the traditional software creation process. The technique used in this dissertation abstracts the level of application for Branch Function Obfuscation to a program's relocatable assembly code rather than working at the binary executable level. The motivation for this move is two-pronged: first, there is an inherent innaccuracy of disassembly on certain architectures and therefore the results of the rewriting-based identification of a program's blocks (used in [38]) cannot be guaranteed accurate. Second, working on a pre-compiled binary is unnecessary for owners of an application's source code wishing to add disassembly protection into a product as obfuscation can be integrated into a single build process. Additionally, it allows for a simplified tool design working primarily on a textual program representation rather than binary.

When performing obfuscation using the techniques of [38], a recovery of the program's assembly code must be performed prior to any other function. Previous work has accomplished this by enforcing input binaries to be compiled with relocation information included. This relocation information, which specifically contains information describing how to access and/or modify binary file section contents [52], is used to aid the recovery of the original assembly code through static disassembly (and is integrated into the obfuscation process). This has two specific problems associated with it; first, the relocation information introduces excessive bloat into a binary file. Second, even with the relocation information present, the disassembly process for producing assembly code has an accuracy of 99.8% [47]. While a highly accurate number, there is a small chance that obfuscation will fail to be performed on the actual assembly code of a software program with this approach.

By abstracting up to the relocatable assembly (or, assembly with no direct addresses, just label references) level of compilation, this small chance of error is eliminated and it can be guaranteed that the original source of a program is obfuscated. Additionally, protecting executables from disassembly during the compilation process is a more natural and transparent means of obfuscation. The special compilation steps and file formats required by previous approaches limit its applicability to real-world solutions.

The primary drawback to this abstraction is that no execution profiling can be applied prior to obfuscation. This profiling is used in [38] to ensure that costly obfuscation is not applied to frequently executed regions of code. However, the goal of improving [38] as presented in Chapter 3 is to maximize protection and minimize cost. By limiting obfuscation to regions of code that are not frequently executed, large gaps between protected code are generated. Additionally, there is no way to ensure that the core algorithms for protection are not within these unprotected regions of code. If this is the case, then the regions protected in [38] are of little importance to the software application as a whole. To reduce the cost of obfuscating frequently executed code regions a lightweight branch function design was developed in Chapter 3. As shown in Section 4.3 the overall cost of obfuscation is sufficiently minimal given its potency, which is accomplished directly through the elimination of disassembly for original code recovery.

4.2.1.2. Two-Stage Design

This tool follows a two-stage design illustrated in Figure 5.2. The initial stage applies branch function insertion to the assembly code and inserts some information

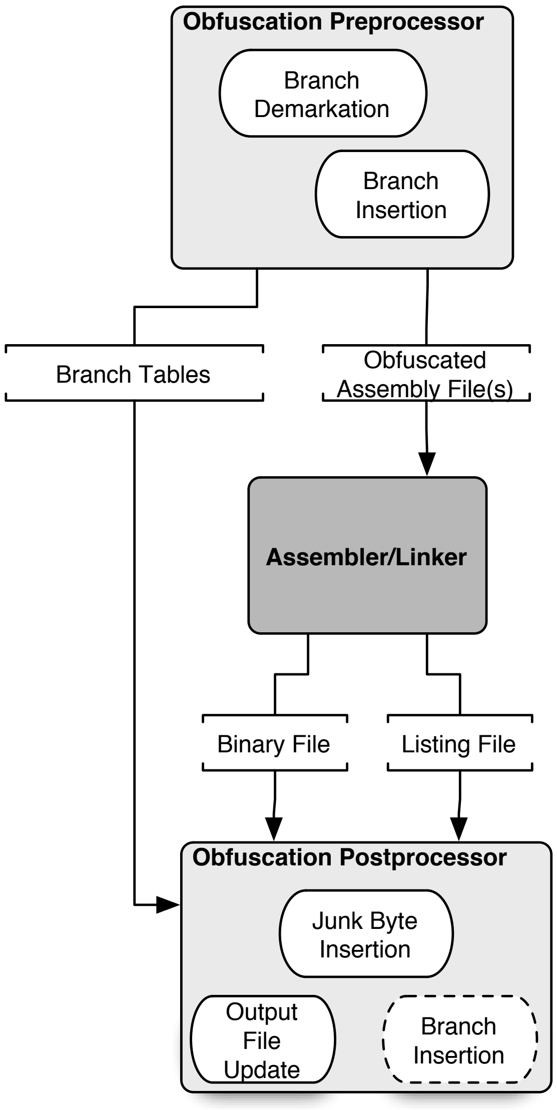


FIGURE 4.2. High-Level ASM for Nebbiolo

into the code for address maintenance. The final stage inserts binary junk bytes into the code section as well as any further branch function insertions dictated by the operation mode. Additionally, it utilizes the inserted utility code to calculate the final branch function target addresses.

The motivation for this two-phase design is due to the fact that there is constraint in the level of obfuscation that can be applied directly to the assembly-language representation of a program. Simple code transformations to the assembly can be applied and structural information about a program can be gathered during the initial phase, but binary data cannot be accurately inserted (*i.e.*, Junk Bytes) without information about final program addresses. The assembly and linking stage further obscures the information gathered from the first stage and the postprocesser relies on unique information inserted during the initial stage into the program in order to maintain address integrity. Therefore, the obfuscation process is distributed across two stages of a program's compilation lifecycle.

4.2.1.3. Modes of Operation

A benefit of working at the abstracted level of relocatable assembly code is the flexibility in how obfuscation can be applied. The mode of operation specifies how much information to consider a candidate for obfuscation. This tool supports, but is not limited, to three modes of application: *Normal*, *Complete* and *Selective*.

Normal Mode specifies application similar to the approach of [38] in that obfuscation is limited to existing candidate blocks of code. Obfuscation is applied to each supported branch instruction (which is the entire set given the techniques of Chapter 3) for the entire body of assembly code as shown in Figure 4.3. This mode is limited by the fact that it does not provide complete code coverage and produces windows of unprotected disassembly while providing a good average level

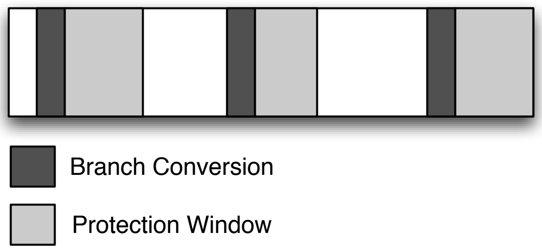


FIGURE 4.3. Code Coverage for Normal Operation Mode

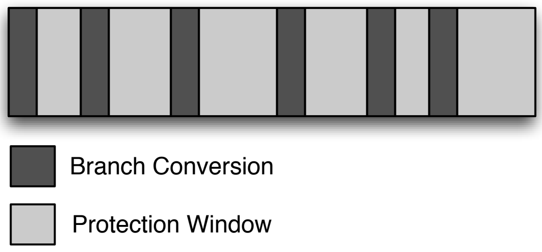


FIGURE 4.4. Code Coverage for Complete Operation Mode

of security.

The design of a lightweight branch function in Chapter 3 was motivated by a desire to provide complete code coverage against disassembly. *Complete* Mode specifies the obfuscation of a source file such that all original code is protected against disassembly as illustrated in Figure 4.4. Complete Mode is an extension of Normal Mode, where a branch function call is inserted at the end of a protected code window to extend the code coverage up to the next window of protected code; that is, broken disassembly alignment should be maintained up to the location of the next sequential junk bytes. This approach provides the maximum level of security at the maximum performance cost. As shown in Chapter 3, its realization does not provide total code coverage, but leaks small windows of code on *undesirable* code regions. Alleviating this is a topic for future consideration.

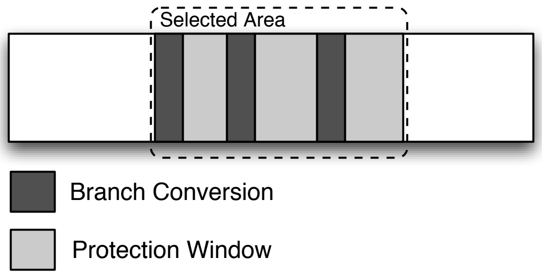


FIGURE 4.5. Code Coverage for Selective Operation Mode

A unique protection against disassembly is available when working at the assembly code level in the compilation process and is provided by the selective mode of operation. *Selective* Mode allows for the limited application of obfuscation specified to discrete regions of code as specified by a developer. That is, selective mode operates by identifying directives within assembly code and applying Complete Mode Obfuscation to individual blocks of code as illustrated in Figure 4.5. This approach provides the maximum level of security for individual functions or algorithms while providing no security to code outside of specified blocks and is useful in situations when intellectual property is limited to small code sections and it is unnecessary to take the performance drawback of further protection.

4.2.1.4. Branch Target Integrity

The primary limitation when working at a higher code level for the protection of a binary file is the information loss during the linking stage of compilation immediately following assembly into an object file. This loss is exacerbated in situations when a source project consists of multiple files. Because of the reliance of disassembly-targeting obfuscation transforms on known, defined addresses within a process memory image it is important for us to ensure that these locations are known and accurate.

To circumvent this information loss a combination of information tables and *demarcation instructions* are used to store relevant code addresses such as addresses of inserted branch function calls and junk bytes. A demarcation instruction is a simple concept that was developed in which a functionally useless but structurally important instruction is inserted at relevant locations within an assembly file in order to act much like a map demarcation point. These instructions allow one to locate certain instruction locations within the final binary executable and are necessary for the abstraction of application level. A typical demarcation instruction sequence is structured as follows:

mov *immediate*, *free_register* // **demarcation**

The value of *immediate* is a unique number randomly generated and associated with a specific location in the source program. No two demarcation instructions will have the same immediate value, and the demarcation value must not be present within a source program. The level of demarcation that can be applied to an assembly program is equivalent to the size of the set of numbers used for *immediate*, and is in realization $\lfloor \frac{2^{32}-1}{2} \rfloor$ due to the use of 32-bit registers on x86.

Because this approach works at the assembly level to apply branch function obfuscation, and because branch function target addresses will change with the insertion of junk bytes during postprocessing and linking, there must be some method by which the integrity of these addresses is maintained. This is accomplished not by archiving addresses during the obfuscation process, but by archiving demarcation IDs into address maintenance tables. The actual branch target and insertions will be located at a static offset from the demarcation instruction. The linking stage will likely rearrange function structure as well as insert functions from other object files and this method of address archiving through demarcation will maintain validity throughout the entire process. To eliminate any chance of information leakage through demarcation, the instruction is removed during postprocessing.

4.2.2. Algorithm

The formal algorithm for implementing improved branch function obfuscation is a straightforward description of how methods developed in Chapter 3 are realized at an abstract level. This algorithm is a technique for obfuscating low-level software against malicious disassembly and control-flow analysis through the transformation of its basic branch structures and insertion of garbage information. It follows two distinct stages, preprocessing and postprocessing, as described below.

Algorithm 3: Improved Branch Function Obfuscation

Preprocess

Take as input $\overline{A} = \{A_1, A_2, ..., A_n\}$ where each A_i is a source file of the form $A_i = \{a^i_1, a^i_2, ..., a^i_m\}$. Additionally, $T = \{t_1, t_2, ..., t_b\}$ is given where each t_i represents a supported branch instruction. An empty list D is used to store demarcation values. The algorithm proceeds as follows:

```
j = 1;
k = 1;
while (j <= n) {
    while (k <= m) {
        if ($a^j_k \in T$) {
            generate unique demarcation $d$;
            insert lightweight branch function prior to $a^j_k$;
            insert demarcation $d$ after branch function;
            add demarcation $d$ to list $D$;
        }
        k += 1;
    }
    k = 1;
    j += 1;
}
```

Postprocess

Given as input is a binary file representing the compiled version of $\overline{A}$, $D = \{d_1, d_2, ..., d_n\}$ representing demarcation points and $J = \{j_1, j_2, ..., j_k\}$ representing each partial junk instruction. The algorithm proceeds as follows:

```
i = 0;
while (i < n) {
    i += 1;
    c = address of $d_i$;
    while (c < address of $d_{i+1}$) {
        L = 0;
        if (c $\neq$ address of $d_i$) {
            insert branch function $B$ at c;
            c = c + sizeof($B$);
        }
        for each (k in J) {
            W = length of protection window for j of length k;
            if (W > L) {
                L = W;
            }
        }
        c = c + L;
    }
}
```

The outer loop in the preprocessing stage forces the evaluation of n source files, where each source file contains m lines of code. This generates an average run-time complexity of $O(n \cdot m)$. The time requirement for the steps contained within the inner-loop require constant time and fall from the complexity of the algorithm. The postprocessing stage again contains an inner- and outer-loop. The outer loop forces a complexity of $O(n)$ as at least n locations must be visited. The inner-loop will only repeat if the length L of protection does not cover all instructions to the next d_i. Its worst case running time occurs when the loop must be run for every instruction between d_i and d_{i+1} which would be some value z contributing to a worst-case complexity of $\Theta(n \cdot z)$. On average, this inner loop will only be required *some* of the time, but the overall average complexity remains $O(n \cdot z)$. The algorithm as a whole is composed of two iterative stages each requiring $O(n \cdot m)$ time on average. It is a very abstract representation of the actual process, which is presented in the next section.[2] This is the first formalized algorithm for Branch Function Obfuscation. The tool Nebbiolo implements a version of this algorithm as described in the next section.

4.2.3. Implementation

Using the above design considerations and formal algorithm, this section gives an abstract view of the actual implementation of our improved obfuscation techniques. The tool is split into three major phases as illustrated in the following sections.

4.2.3.1. Preprocessing Phase

The *Preprocessing Phase* is responsible for applying simple obfuscating transforms on a source program, including the insertion of lightweight branch functions on control-altering assembly instructions as well as gathering information about the source-level structure of a program. Primarily, its job is to insert protecting branch function calls prior to each branch and archive the insertion points through demar-

[2]For example, this algorithm makes no mention of the offsets generated by code insertion which have the potential to break references to immediate addresses within a binary file.

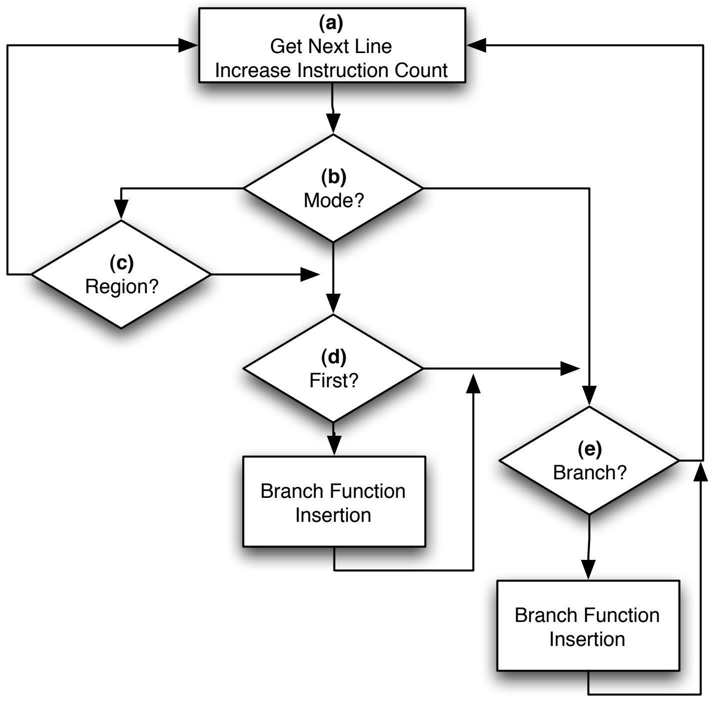

(a)
Get Next Line
Increase Instruction Count
(b)
Mode?
(c)
Region?
(d)
First?
(e)
Branch?
Branch Function Insertion
Branch Function Insertion

FIGURE 4.6. Abstract Implementation of Preprocessing Stage

cation instructions. It is infeasible to insert junk bytes or additional lightweight branch function calls (as required by the complete operation mode) at this point in obfuscation due to the dependance on final machine code for calculating their size and effectiveness. The preprocessing phase is illustrated in Figure 4.6.

From an algorithmic view, the preprocessing stage follows several simple steps of operation based on the chosen mode. Its basic goal is to parse through a source file and perform obfuscating transformations based on some discernible semantics within the code. At (a), the preprocessor simply parses the next line of code. There is no functionality for enforcing source conventions as this is handled later by the assembler. Once a single line of code has been gathered, a path of execution is chosen at (b) based on the selected operation mode. At this point the flow of control essentially mimics a waterfall based on the path chosen. If selected

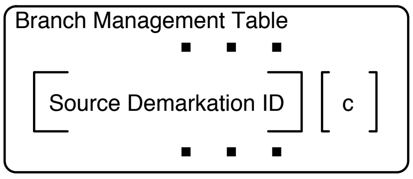


FIGURE 4.7. Branch Management Table Structure

mode is specified, then at (c) the preprocessor will check if the current instruction is in a predefined protection region. This notion of being in or out of a region is defined in the assembly code by directives that indicate the head and tail of a region where protection is desirable. If the current instruction is not in a specified region of protection and selective mode is specified then control returns to (a). If not, then at (d) a check is made if this is the first instruction in the selected block of code. If at (b) the operation mode was set to complete, then control proceeds to this location. In a situation where one wants to completely protect a region or body of code, the first location must always contain an inserted branch function as handled at this location in Figure 4.6. If the current instruction is not the first in a block or if the operation mode was set to normal then control will continue at (e). If the current instruction is the first in a block of code and the relevant operation mode was specified control will then insert a branch function before proceeding to (e). The core functionality of the preprocessing stage is performed at (e) in that if the current instruction is a valid branch instruction then a lightweight branch function call is inserted, the relevant table information is created and a demarcation is inserted. If this instruction is not a valid branch, control returns to (a). This process is performed on the body of an assembly file provided as input to the preprocessor. Once all files have been exhausted a file is generated giving a definition for the branch function(s).

One of the most important stages of the preprocessor is in the construction of information tables representing structure and transformations applied to a program's source. These tables are similar to those gathered during the first pass of a traditional two-pass assembler. Using the structure shown in Figure 4.7, entries are created for each branch insertion as they are performed during obfuscation. A demarcation instruction is inserted following a branch function call. An entry is created in the tables for each inserted branch function call with their corresponding demarcation as well as branch type (as different call insertions and branch functions can be used with different branch types). This information is gathered in a top-down fashion. These tables are utilized in the postprocessing stage for the insertion of junk bytes and the computation of offset values for address translation. An additional table is created during preprocessing which indicates "protection profiles", or regions of code specified for protection.

As previously stated, there is a limitation on the degree of obfuscation applied in the preprocessing stage. That is, due to the operation at an abstracted level of code in which obfuscation is unable to accurately insert the additional lightweight branch function calls as required by selective and complete modes of operation. This is justified by the fact that boundaries of protection windows generated by junk byte insertion cannot be discerned until the postprocessing stage when instruction addresses and structure are formalized.

4.2.3.2. Assembly and Linking

Assembly and linking is an intermediary stage, and the actual obfuscation process "wraps" around it. For the purposes of this tool (developed for Linux/x86) the compiler *gcc* is used, as it is the de facto standard for this operating environment. It is invoked from within the program with the necessary input parameters.

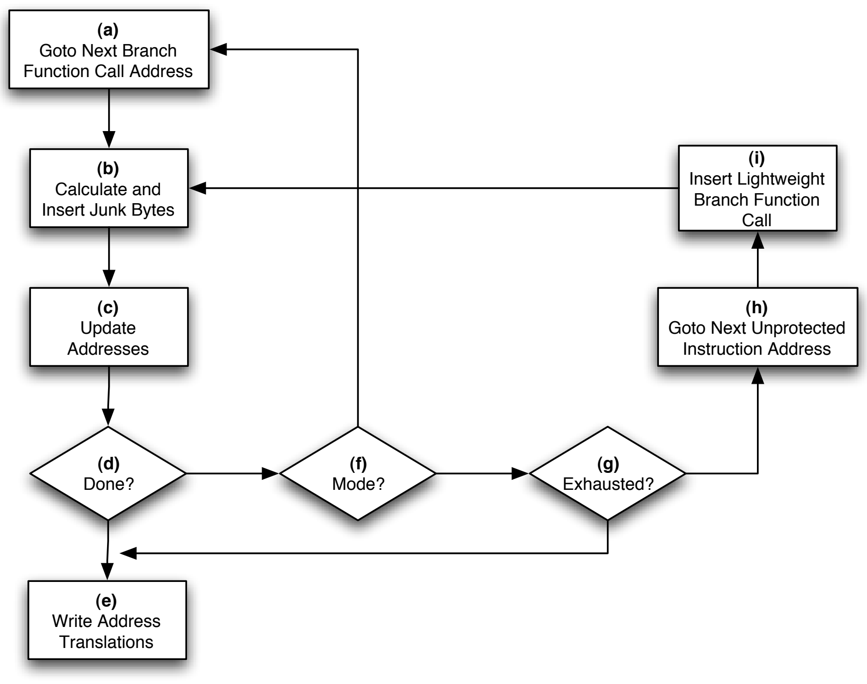

(a)
Goto Next Branch Function Call Address
(b)
Calculate and Insert Junk Bytes
(c)
Update Addresses
(d)
Done?
(e)
Write Address Translations
(f)
Mode?
(g)
Exhausted?
(h)
Goto Next Unprotected Instruction Address
(i)
Insert Lightweight Branch Function Call

FIGURE 4.8. Abstract Implementation of Postprocessing Stage

4.2.3.3. Postprocessing Phase

The *Postprocessing Phase* is responsible for the insertion of junk bytes into a binary file, the insertion of additional branch function calls for complete mode obfuscation and the generation of the final binary image. A general view of the postprocessing stage is given in Figure 4.8.

Again, the postprocessing phase follows several discrete and logical steps of operation. First, the binary file is read as input to the postprocessing stage and a listing file is generated to give information about addresses within the binary. The algorithm proceeds at (a) to go to the next branch function call as indicated within the branch management tables. At (b) junk bytes are inserted based on the algorithms described in previous sections; parameters for the branch function are updated and the offset generated is tabulated into a file giving the total offset for a given code region at (c). At (d) a check is made to see if the set of branch functions has been exhausted or the end of code has been reached. If so, then at (e) the final binary image is generated by updating address dependancies. Note that prior to this the binary file must be in its final form. At (f) a check is made on the operation mode. If Normal operation mode is chosen, then control returns to (a); if selective or complete operation mode is chosen then the algorithm proceeds to (g). At (g), a check is made indicating whether the current code region flagged for protection is exhausted or fully protected. If true, then control proceeds to (d) and ultimately if selective mode is chosen then at (f) will be redirected to (a) at the next selected region; in the case of complete mode the protection region is the entire body of code. If the check at (g) is false then the algorithm proceeds to the next unprotected instruction address given by (b). At (i) a lightweight branch function is inserted and the relevant table information is created. After (i) the process continues at (b) in the regular algorithmic steps.

The justification for a postprocessing phase is two-fold. First, in order to accurately compute the maximal number of junk bytes to insert after an extended branch function call one must first know the final binary structure of the basic block of code following the point of insertion. That is, any changes in the binary structure of these instructions will alter the point at which disassembly realignment occurs. This forces one to insert these bytes at a time when it is considered *safe* to simulate disassembly and compute the number of bytes to insert. Second, in order to achieve the level of protection provided by the complete mode of operation the boundaries of protection provided by a normal mode of operation must be known. This can only be determined by calculating and inserting junk bytes, which require the final binary structure utilized during the postprocessing stage.

The postprocessing stage, and thus this protection tool, completes through the updating of address-dependent instructions in the binary file. A direct scan is made of the listing file to locate addresses of these instructions and, using the information in the offsets table, the instructions are found and updated to reflect the total offset of their target's locale. Once this is complete the final binary is fully protected.

4.3. Experimental Evaluation

The goal in creating an effective tool design is to minimize the time and space overhead generated through obfuscating transforms while providing a high degree of potency against attack. The results given in Chapters 2 and 3 indicate the efficacy of proposed obfuscating transforms against target disassembly algorithms. In addition to achieving high levels of protection, it is important to minimize the overall time and space overhead required for protection application. The tools presented in this chapter provide significant engineering optimizations for the development of effective, efficient methods of software protection.

4.3.1. Tempranillo

As previously stated while developing a formal method of evaluation, the route chosen for generating results in this dissertation is through the obfuscation and disassembly of the SPEC benchmark suite. These benchmark applications are compiled using gcc version egcs-2.91.66 with optimization level -O3. For this simulation the SPECint2000 suite is preferred over the previously used SPECint95.

Structural Obfuscation is applied first using the Tempranillo tool; the applications are then processed using the methods of Linn and Debray with their tool PLTO [46], as optimal results occur when the two are used in conjunction

TABLE 4.1. No. of Instructions in Final Binary Executable

Program	**Original**	**Linn/Debray**	**Structural**
bzip2	333,249	687,989	690,509
crafty	443,632	855,406	858,268
gap	658,808	1,350,104	1,363,866
gzip	333,831	692,409	694,046
mcf	285,720	597,242	597,355
parser	379,023	780,588	786,119
vortex	672,267	1,371,189	1,371,373

for both disassembly accuracy and time requirements. To represent the original version of each tool the files were compiled as above and not obfuscated but simply written into a final executable with obfuscation disabled using the tool in [38]. Each application was tested against the disassembler developed by Kruegel *et al.* in [35]. All results were obtained using an average of ten tests for each benchmark utility on a 1.0 GHz Celeron with 512MB RAM running SuSE Linux.

Space requirements, in addition to those of [38], are given in Tables 4.1 and 4.2. Table 4.1 gives the additional number of instructions present in the final binary while Table 4.2 gives the increase in the size of code in the binary after obfuscation. It can be readily discerned from these two tables that the storage overhead generated by this tool is small in comparison to other binary obfuscation tools giving an average increase in the number of instructions to be 0.44% and the total increase in physical size for a binary code section of 0.36%. Note that the values associated with the **Structural** column indicate a final binary file in which both Structural Obfuscation and the techniques of Linn and Debray have been used.

TABLE 4.2. Size in Bytes of *.text* Section in Final Binary Executable

Program	**Original**	**Linn/Debray**	**Structural**
bzip2	101,715	232,009	232,348
crafty	129,711	281,128	281,482
gap	204,988	457,659	461,074
gzip	102,507	234,177	234,527
mcf	89,110	203,656	203,761
parser	119,191	266,553	267,890
vortex	213,949	471,785	475,455

Execution requirements are given in Table 4.3, and are structured in the same manner as previous tables. Again, it can be observed that a minimal time overhead is associated with structural obfuscation applied in conjunction with PLTO. Though a first realization of Structural Obfuscation, this tool illustrates that a small, yet statistically significant, measure of cost (2.86%) is associated with these protections while maintaining high potency against attack (as shown in [55]). It is important to note that the execution overhead of Linn/Debray in Table 4.3 does not illustrate the high overhead given in [38]. While obfuscation was performed with the default parameters given in their tool, their excessive run-time overhead was not reproduced through our tests. One possible explanation for this is that more aggressive obfuscation was applied for their work, thus introducing further computational overhead. However, this is insignificant, as the results of this section illustrate the overhead of Structural Obfuscation beyond [38] when used in conjunction.

As shown in Figures 4.4 and 4.5, the relative cost associated with this new transform is small when compared to other *disassembly-specific* obfuscating techniques. Given its high potency when specifically targeting disassembly ex-

TABLE 4.3. Execution Time in Seconds

Program	**Original**	**Linn/Debray**	**Structural**
bzip2	17.0	17.6	17.2
crafty	6.30	7.20	7.67
gzip	5.7	6.4	6.5
mcf	∼1	1	1.1
parser	11.1	12.3	12.4
vortex	12.0	62.8	64.1

ecution requirements, it gives a very desirable ratio of Potency versus Cost as introduced in [19] (though the potency derives from an entirely new application). Given that it is the first such technique to target disassembly time, these results regarding obfuscation cost make this method of protection simple and cheap to use in conjunction with other obfuscating transforms.

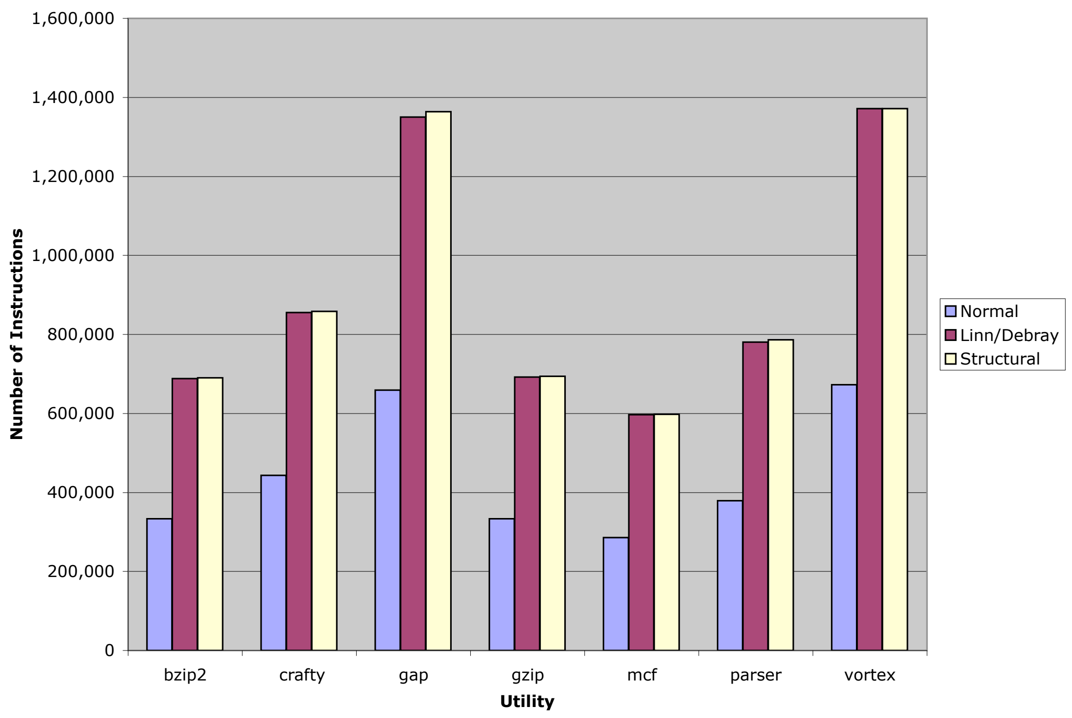


FIGURE 4.9. Instruction Overhead for Tempranillo

4.3.2. Nebbiolo

Nebbiolo was developed as an improvement over existing obfuscation tools targeting disassembly as specified by design decisions in this chapter. Specifically, its realization was designed such that the overall overhead associated with obfuscation would have minimal impact on the final binary image. As illustrated in this section, the final results concerning resource overhead of obfuscation provide a much more optimal and desirable pattern than existing techniques, leading to the best Potency versus Cost relationship in current literature. All results were obtained on a Dual Xeon 3.06GHz machine with 2GB RAM running RedHat Enterprise 3.

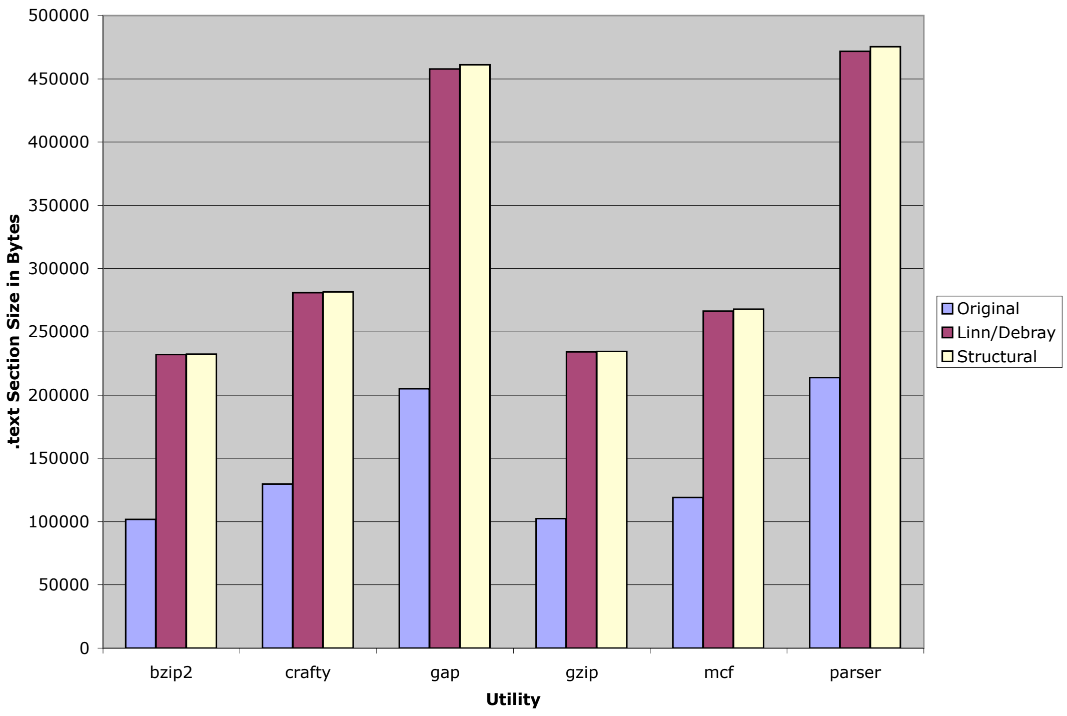


FIGURE 4.10. .text Section Overhead for Tempranillo

The output file sizes associated with our work are a steady improvement over existing tools as illustrated in Table 4.4. Each obfuscated value was compared with the original (via the computation of $\frac{Obfuscated}{Original}$) to determine a percentage increase in resource requirements. The average increase in resource consumption for Linn and Debray over the seven benchmark tools was 1,716.42%, leading to a costly obfuscation process. This can be directly associated to their requirement for input files to have relocation information compiled in for their preliminary code recovery step as well as the heavyweight branch function code. To alleviate this, our tool abstracted obfuscation to the assembly level to provide a streamlined obfuscation process which paid careful attention to the additional overhead generated. As a result, the final obfuscated binary files using our techniques were analyzed in the same manner mentioned above to determine an average storage increase of 30.87% over the original binary file. These numbers indicate that the

TABLE 4.4. Size in Bytes of Final Binary Executable

Program	**Original**	**Linn/Debray**	**Nebbiolo**
bzip2	52,837	1,132,000	70,939
crafty	236,981	1,402,392	276,955
gap	458,926	2,119,196	590,096
gzip	56,794	1,144,288	75,413
mcf	18,224	996,832	22,893
parser	138,659	1,279,456	209,024
vortex	545,762	2,246,112	694,419

approach given in this chapter for realization of *Lightweight* Branch Function Obfuscation alleviates some overhead associated with [38] and provides a very low total cost of obfuscation regarding space.

Execution overhead was a primary motivation for the design of a lightweight branch function in Chapter 3. Due to the rigorous application of obfuscating transforms during the protection process, a lightweight function is required to minimize the final run-time requirements for execution. At the hardware level, any excessive introduction of branch instructions will cause performance degradation due to the inability to speculatively execute instructions and properly pipeline instruction sequences; however, this problem is outside the scope of this research. Though we were unable to reproduce the numbers given in [38] (an average execution-time increase of 52%) it can be assumed that their simulations are correct. Given this, the run-time overhead associated with our techniques seems to be a solid improvement over existing methods as illustrated in Table 4.5. It can be readily observed that the average overhead associated with our methods is 43.14%, which is minimal given the use of a lightweight branch function in conjunction with rigorously applied obfuscation. When comparing the execution overhead of Linn/Debray with our techniques one can observe a greater spread of executions times in [38]. Using Student's *t*-Test we can determine if our lowered average execution time is statistically significant and judge the overall impact of this reduction. Applying this test we observe the following information about the two data sets:

Linn/Debray
Mean = 1.56
95% confidence interval for Mean: 1.326 thru 1.786
Standard Deviation = 0.368
Hi = 2.24 Low = 1.14
Median = 1.54
Average Absolute Deviation from Median = 0.291

Nebbiolo
Mean = 1.43
95% confidence interval for Mean: 1.185 thru 1.678
Standard Deviation = 0.196
Hi = 1.72 Low = 1.12
Median = 1.46
Average Absolute Deviation from Median = 0.134

The final t-Test results are as follows:

t= 0.800
sdev= 0.301
degrees of freedom = 13
The probability of this result, assuming the null hypothesis, is 0.44

The purpose of the test is to ensure that the measured difference is not due to "chance" and statistically significant. Given the above results we can see that

TABLE 4.5. Execution Time in Microseconds

Program	**Original** (T_O)	**Nebbiolo** (T_N)	**Slowdown** ($\frac{T_N}{T_O}$)
bzip2	4802696	7020101	1.46
crafty	2545435	3220484	1.26
gap	545048	781108	1.43
gzip	1145396	1671935	1.46
mcf	197291	221005	1.12
parser	1546211	2655617	1.72
vortex	4004910	6300925	1.57

the likely ranges for the sample mean overlap and we cannot reject the null hypothesis; thus, the two mean values could *possibly* be the same and we can claim that the difference observed in testing is not statistically significant. Even though the results are not statistically a significant improvement, they still have importance to the overall impact of this work. That is, given a more aggressive level of obfuscation no overall increase in execution time was present. This greater aggression in obfuscation led to significantly lower Instruction Leakage and gave a much more desirable result in regards to potency versus application cost, and provides an improvement over current literature.

The traditional problem of security versus performance is further optimized with our techniques for the protection of binary files against static disassembly. In [38] a good level of protection was given against disassembly but at an average run-time cost of 52%. The results given here indicate a much more desirable ratio of protection versus overhead and are meant to trigger a fundamental shift in the development of protection tools targeting static disassembly. It has been shown through our work that abstracting away from the binary level to assembly allows for more accurate protections to be applied to an executable as well as reduced

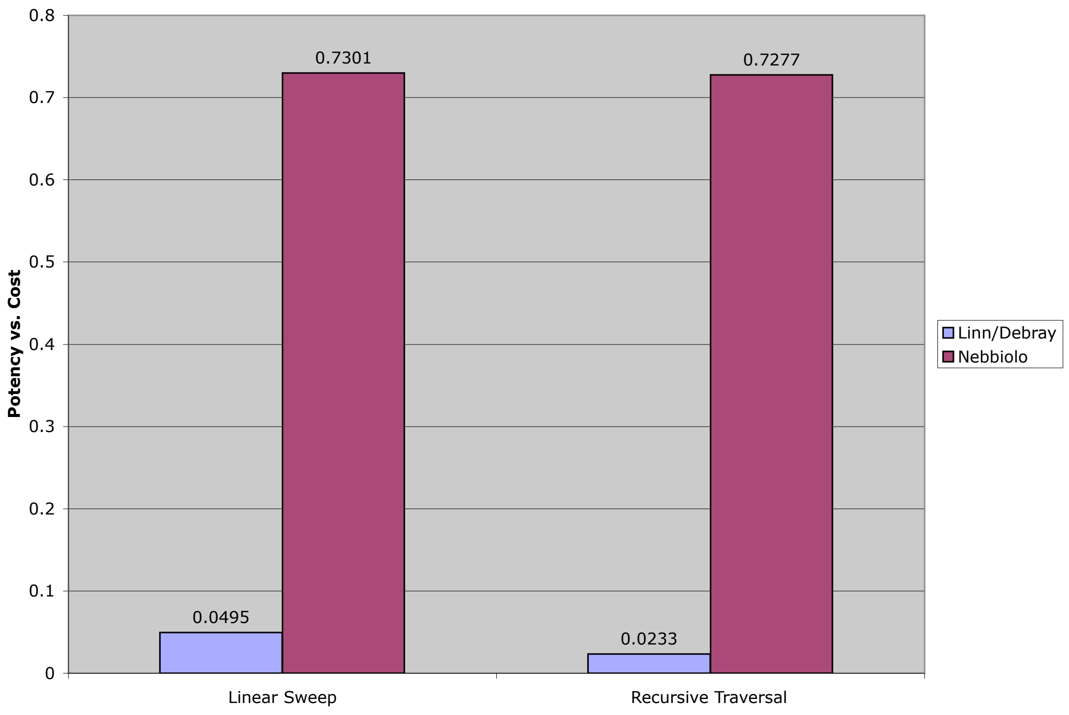


FIGURE 4.11. Potency versus Storage Cost

resource requirements in the protected file while achieving very high levels of protection. The use of a lightweight branch function reduced the impact of more frequent obfuscation on the execution requirements of the protected binary.

Figures 4.11 and 4.12 illustrate graphically the Potency versus Cost ratio for both space and time requirements comparing our work to other existing binary obfuscation techniques [38]. These further illustrate the impact of our work to current literature as well as the optimization of this ratio. The ratio is defined by the following equation:

$$\text{Potency vs. Cost} = \frac{\mathit{100 - IL}}{\mathit{1 + Overhead}}$$

In this equation, the numerator specifies the percent of protected instructions while the denominator represents the actual percentage overhead in resource consumption. The base resource consumption is 100%, signifying no associated over-

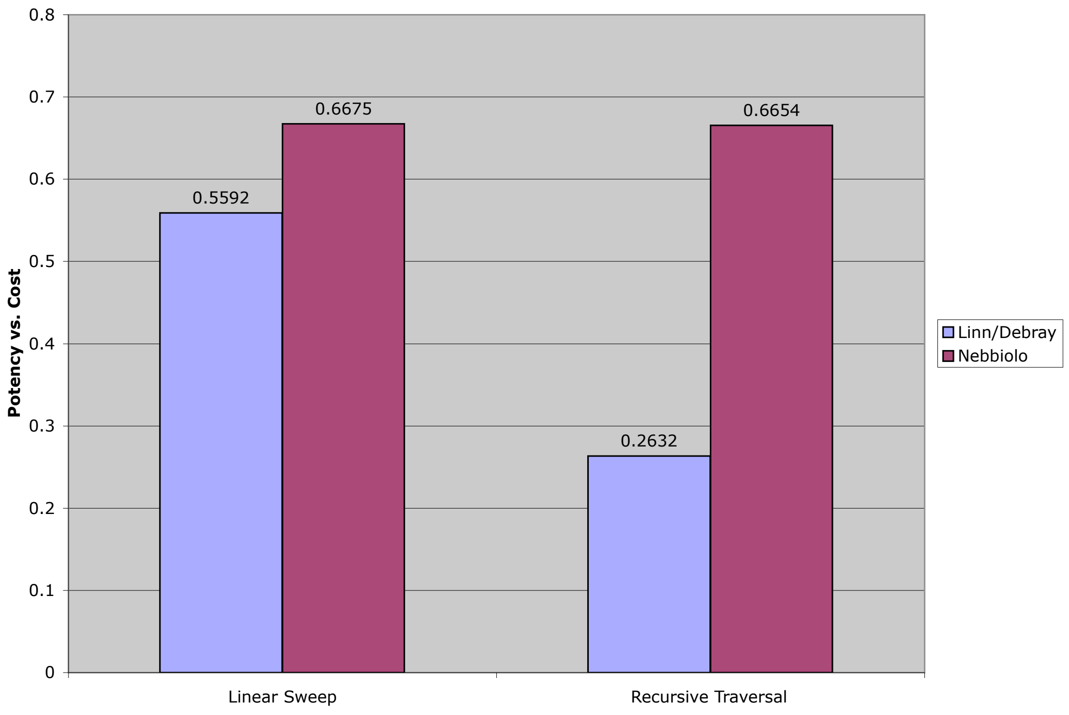


FIGURE 4.12. Potency versus Execution Cost

head. It can be readily observed that as IL and Overhead decrease, the resulting value approaches 1. This result lies in the range $(0, 1]$ where larger values indicate more desirable protection techniques, or a higher Potency versus Cost.

Future work on this topic should be focused on the development of a lightweight branch function that takes into consideration its execution at the hardware level, such that it can be effectively pipelined. This will further reduce the time overhead associated with obfuscation, and produce an even stronger ratio of potency versus cost. Additionally, new methods for the calculation of junk bytes must be developed such that the presence of undesirable code regions (as defined in Chapter 3) are eliminated from the final binary executable. One such approach for this would be to extend the pool of junk bytes used, as the creation of every possible instruction permutation is the only possible way to reduce this problem.

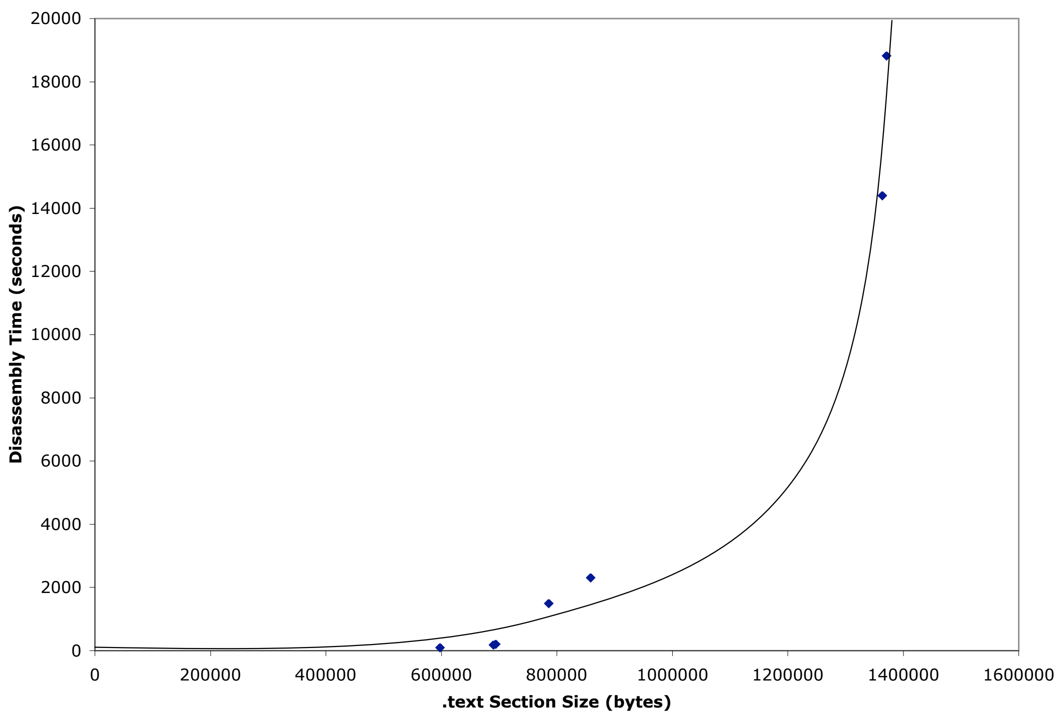


FIGURE 5.1. Disassembly Time Requirements

5. CONCLUSION

The field of software protection against disassembly is very young, and the work of this dissertation represents the most extensive and formal treatment of the subject to date. The contributions of this dissertation achieve sound advancements to the field of disassembly prevention through obfuscation. In Chapter 2 a new classification of obfuscating transform was presented. This Structural Obfuscation Transform was then utilized for the increase in the time requirements of static disassembly. It was the first approach presented to date that addressed not the reduction in information leakage, but the increase in disassembler time. As illustrated in Figure 5.1, proper obfuscation of a binary file with structural transforms had a profoundly adverse impact on disassembly performance.

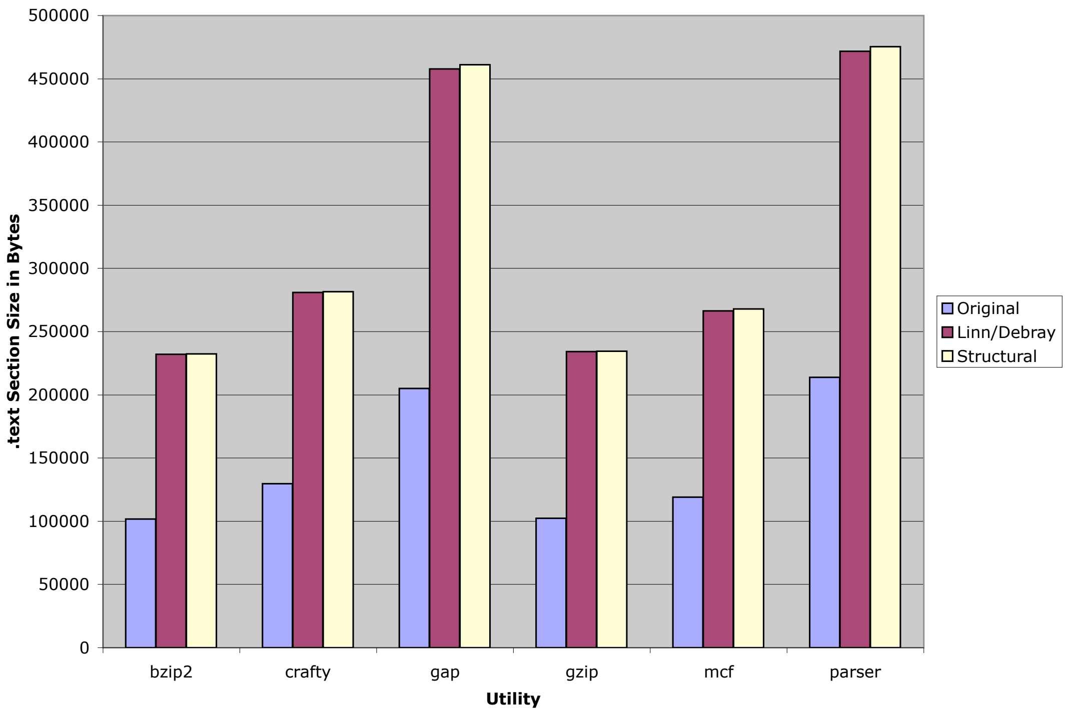


FIGURE 5.2. .text Section Overhead for Tempranillo

This positive protection mechanism gave very strong protection against disassembly in a non-traditional manner, and illustrated very small resource overhead into a binary file. Figure 5.2 illustrates graphically this minimal overhead. It can be readily observed from these results a small measure of cost associated with very positive protection strength.

Chapter 3 presented an approach accomplishing the best protection against static disassembly in current literature. These techniques took fundamental concepts in current literature and refined them, such that much more desirable protection characteristics can be accomplished at a minimal cost to the resultant binary file. Additionally, problems associated with current literature are identified and addressed. Techniques for the theoretically complete protection against static disassembly for binary files are given, and problems with their realization

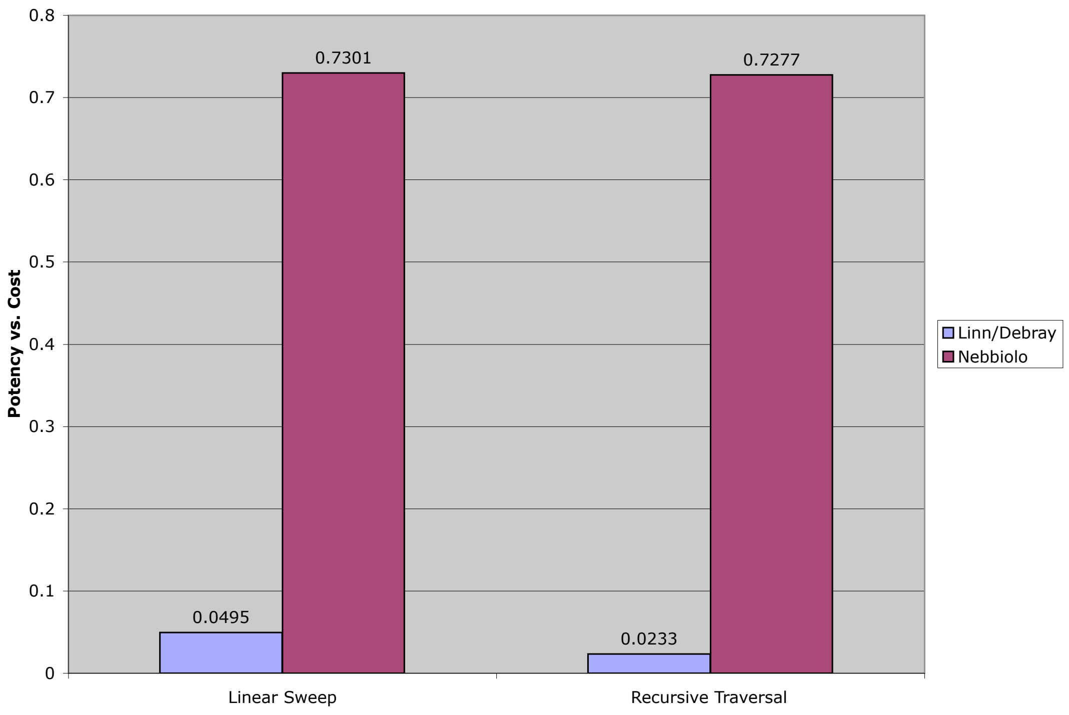


FIGURE 5.3. Potency versus Storage Cost

are explained. The chapter concludes with results illustrating the smallest degree of information leakage (an average of 4.61% between the two disassembly algorithms) to date when compared with the most modern work in this area [38], as well as the definition of Undesirable Code for disassembly protection and some potential approaches for its future solution.

Chapter 4 introduced new engineering techniques for the creation of binary protection tools not susceptible to the overhead associated with modern techniques. Additionally, it presented formal algorithms for the realization of topics covered in Chapters 2 and 3. The first formal design and algorithm for a Structural Obfuscating Transform was given, along with analysis of the required overhead in applying said transform to a binary program. Also presented is a formal algorithm and realization for improved branch function obfuscation, leading

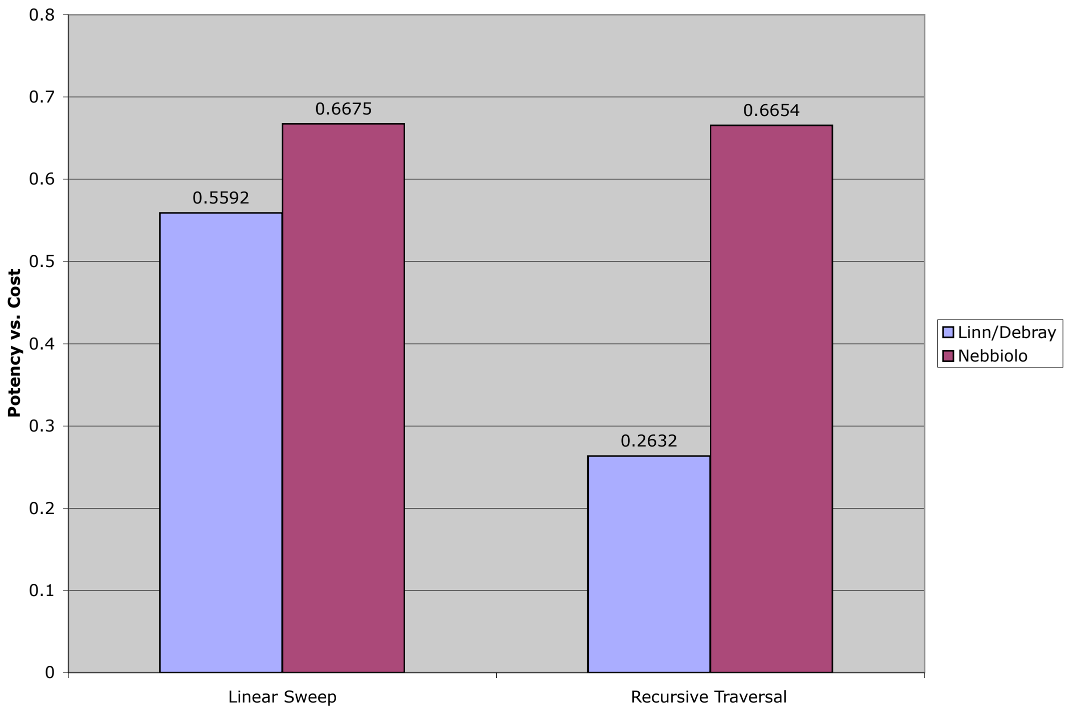


FIGURE 5.4. Potency versus Execution Cost

to the most desirable results in current literature. A formal equation for the computation of Potency vs. Cost is developed, and is used to illustrate the strength of our techniques over previously published methods. The obtained results integrated the low information leakage values from Chapter 3 to analyze the Potency vs. Cost ratio for Improved Branch Function Obfuscation implemented with the techniques of this chapter to achieve values noted in Figures 5.3 and 5.4.

An additional contribution of this dissertation lies within Chapter 1, which presented for the first time a formal method of evaluating the efficacy of protections against static disassembly. It was meant to even the playing field for all future research in this arena, such that the wheel need not be recreated each time. In our work, we developed a concrete approach based on feedback from various sources as well as our own techniques to define a simple, uniform technique for the realization of the information leakage (IL) metric.

Software protection is not a new topic, but it has been gaining interest in recent years. Obfuscation has been more aggressively researched by numerous groups in the past seven years, and its application is expanding each year. The use of obfuscation specifically to target static disassembly is a young and relatively untouched topic. That is, this dissertation presents the most thorough and broad treatment of the topic to date. Prior to this, only one such publication existed [38] directly moving the protection of binary programs against static disassembly from haphazard code modification to a defined, theoretically founded technique.

As this research is still young, much can be done in the future to advance the topic. Specifically identified in previous chapters, developing new branch function designs that take into consideration not only optimal software execution but also the hardware design (such as pre-fetching, pipelining, etc.) are a necessity to increase the Potency vs. Cost metric. Also, a currently unrealizable theoretical model for the complete prevention of static disassembly on x86 was presented. The defined Undesirable Code leading to information leakage can hopefully be solved through the analysis and inclusion of every possible partial instruction combination on the x86 architecture. It will be exciting to see what happens with the newly defined Structural Obfuscation Transform. Its use, to date, has been limited to this dissertation, but it has promise in other areas of software protection. Specifically, it could be possible to utilize this transform for the translation of protection over to dynamic disassembly wherein structures could be used to fool a disassembler into non-terminating loops; however, due to the wide availability of debugging information on modern CPUs and in most operating systems, this could prove infeasible. These topics are the most fundamental areas of future work in the topic of preventing static disassembly through obfuscation, and should be the focus of any further research in this area.

As the world moves forward, and access to computing resources as well as information increases, the need for means by which to securely execute proprietary software on untrusted environments grows in kind. It is this drive that is forcing research in computer and software security to look to areas outside the traditional monolithic model of client-server security, and find new techniques for securing computing environments. It is truly fascinating to observe this from the inside, in the guise of a security researcher; one cannot help but be excited for the years to come, as security becomes more intertwined with the technological world.

BIBLIOGRAPHY

[1] A. V. Aho, R. Sethi and J. D. Ullman. *Compilers: Principles, Techniques And Tools*. Addison-Wesley Publishers. 1986.

[2] D. Aucsmith. Tamper-Resistant Software: An Implementation. In *Proc. of Information Hiding: First International Workshop*, LNCS Volume 1174, pp. 317-333. 1996.

[3] Bastard Disassembler, *libdisasm*: x86 Disassembler Library. http://bastard.sourceforge.net/libdisasm.html

[4] J. Borland. *Controversial piracy legislation gets EU thumbs-up*. Silicon.com. March 10, 2004.
http://www.silicon.com/hardware/storage/0,39024649,39119040,00.htm

[5] P. Cerven. *Crackproof Your Software*. San Francisco: No Starch Press. 2003.

[6] C. Cifuentes. *Reverse Compilation Techniques*. Ph.D. Dissertation, Queensland University of Technology, School of Computing Science. July 1994.

[7] C. Cifuentes. Interprocedural Data-Flow Decompilation. *Journal of Programming Languages*, Volume 4 No. 2, pp. 77-99. 1996.

[8] C. Cifuentes and K. J. Gough. A Methodology for Decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Informtica*, pp. 257-266, Buenos Aires, Argentina. August 1993.

[9] C. Cifuentes and K. J. Gough. Decompilation of Binary Programs. *Software - Practice and Experience*, 25(7): 811-829. July 1995.

[10] C. Cifuentes and M. Van Emmerik. UQBT: Adaptable Binary Translation at Low Cost. *IEEE Computer*, 33(3): 60-66. March 2000.

[11] C. Cifuentes and M. Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. *Science of Computer Programming*, 40(2-3): 171-188. July 2001.

[12] C. Cifuentes, D. Simon and A. Fraboulet. Assembly to high-level language translation. In *Proc. of the International Conference on Software Maintenance*, pp 228-237. November 1998.

[13] C. Cifuentes, M. Van Emmerik, Ung, Simon and Washington. Preliminary Experiences with the Use of the UQBT Binary Translation Framework. In *Proc. of Workshop on Binary Translation*, Technical Committee on Computer Architecture News, IEEE CS Press, pp. 12-22. 1999.

[14] C. Cifuentes, T. Waddington and M. Van Emmerik. Computer Security Analysis through Decompilation and High-Level Debugging. In *Proc. of Eighth Working Conference on Reverse Engineering (WCRE 01)*, pp. 375-380. 2001.

[15] F. B. Cohen. Operating System Protection Through Program Evolution. 1992. http://all.net/books/IP/evolve.html

[16] R. Cohn, D. Goodwin. P. G. Lowney and N. Rubin. Optimizing Alpha Executables on Windows NT with Spike. *Digital Technical Journal*, Vol. 9, No. 4, pp. 3-20. 1997.

[17] C. Collberg and C. Thomborson. Software Watermarking: Models and Dynamic Embeddings. In *Proc. of 26th ACM Symposium on Principles of Programming Languages (POPL 1999)*, pp. 311-324. January 1999.

[18] C. Collberg and C. Thomborson. Watermarking, Tamper-Proofing and Obfuscation - Tools for Software Protection. Technical Report TR00-03, The Department of Computer Science, University of Arizona. February 2000.

[19] C. Collberg, C. Thomborson and D. Low. A Taxonomy of Obfuscating Transforms. Technical Report 148, Department of Computer Sciences, The University of Auckland. July 1997.

[20] C. Collberg, C. Thomborson and D. Low. Breaking Abstractions and Unstructuring Data Structures. In *Proc. of 1998 IEEE International Conference on Computer Languages*, pp. 28-38.

[21] C. Collberg, C. Thomborson and D. Low. Manufacturing Cheap, Resilient and Stealthy Opaque Constructs. In *Proc. of 25th ACM Symposium on Principles of Programming Languages (POPL 1998)*, pp. 184-196. January 1998.

[22] The Digital Millennium Copyright Act of 1998. http://www.copyright.gov/legislation/dmca.pdf

[23] S. Drape, O. de Moor and G. Sittampalam. Transforming the .NET Intermediate Language Using Path Logic Programming. In *Proc. of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pp. 133-144. 2002.

[24] R. El-Khalil and A. D. Keromytis. Hydan: Hiding Information in Program Binaries. In *Proc. of 6th International Conference on Information and Communications Security (ICICS)*. 2004.

[25] P. Falcarin, M. Baldi and D. Mazzocchi. Software Tampering Detection using AOP and Mobile Code. http://citeseer.ist.psu.edu/697036.html

[26] A. Fraboulet, G. Hurard and A. Mignotte. Loop alignment for memory accesses optimization. In *Proc. of the Twelfth International Symposium on System Synthesis*, Boca Raton, Florida. November 1999.

[27] M. L. Fredman, J. Komlos and E. Szemeredi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM*, 31(3):538-544. July 1984.

[28] GNU Project - Free Software Foundation, *objdump*, *GNU Manuals Online*, http://www.gnu.org/software/binutils/manual/html_chapter/binutils_4.html

[29] GNU Project - Free Software Foundation, *strip*, *GNU Manuals Online*, http://www.gnu.org/software/binutils/manual/html_chapter/binutils_8.html #SEC10

[30] B. Horne, L. Matheson, C. Sheehan and R. E. Tarjan. Dynamic self-checking techniques for improved tamper-resistance. In *Proc. of Workshop on Security and Privacy in Digital Rights Management.* 2001.

[31] W. C. Hsieh, D. Engler, and G. Back. Reverse-Engineering Instruction Encodings. In *Proc. of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts. June 2001.

[32] ISLDasm - The ISL Disassembler. http://www.engr.orst.edu/ vandyke/html/isldasm.html.

[33] K. Kaspersky. *Hacker Disassembling Uncovered: Powerful Techniques To Safeguard Your Programming.* Wayne: A-List Publishing LLC. 2003.

[34] J. Koziol, D. Litchfield., D. Aitel, C. Anley, S. Eren, N. Mehta and R. Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes.* Indianapolis: Wiley Publishing, Inc. 2004.

[35] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of USENIX Security 2004*, pp. 255-270. San Diego, CA. August 2004.

[36] J. Larus and T. Ball. Rewriting executable files to measure program behavior. In *Software - Practice and Experience*, 24(2):197-218. February 1994.

[37] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proc. of 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pp. 168-177. November 2000.

[38] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proc. of 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pp. 290-299. October 2003.

[39] M. Ludwig. *The Giant Black Book Of Computer Viruses*. Show Low: American Eagle Publications. 1995.

[40] R. Muth, S. Debray, S. Watterson and K. De Bosschere. ALTO: A Link-Time Optimizer for the Compaq Alpha. *Software - Practice and Experience*, 31:67-101. January 2001.

[41] A. Mycroft. Type-Based Decompilation. In *Proc. of European Symposium On Programming (ESOP 99)*. 1999.

[42] A. Mycroft, A. Ohori and S. Katsumata. Comparing Type-Based and Proof-Directed Decompilation. In *Proc. of the Eighth Working Conference on Reverse Engineering (WCRE 01)*, pp. 362-367. 2001.

[43] J. Nagra, C. Thomborson and C. Collberg. A Functional Taxonomy for Software Watermarking. In *Conferences in Research and Practice in Information Technology*, Vol. 4. 2002.

[44] T. Ogiso, Y. Sakabe, M. Soshi and A. Miyaji. Software Obfuscation on a Theoretical Basis and its Implementation. *IEEE Trans. Fundamentals*, E86-A(1). January 2003.

[45] Sarovar.org - Playfair DRM Stripper for Apple AAC Audio Files (Taken down due to legal notice from Apple Computer Inc.). April 15, 2004. http://sarovar.org/forum/forum.php?forum_id=474

[46] B. Schwarz, S. Debray and G. R. Andrews. PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In *Proc. of 2001 Workshop on Binary Translation (WBT-2001)*. 2001.

[47] B. Schwarz, S. Debray and G. R. Andrews. Disassembly of Executable Code Revisited. In *Proc. of IEEE 2002 Working Conference on Reverse Engineering (WCRE)*, pp. 45-54. October 2002.

[48] R. L. Sites, . Chernoff, M. B. Kirk, M. P. Marks and S. G. Robinson. Binary Translation. *Communications of the ACM*, 36(2): 69-81. February 1993.

[49] A. Srivastava and D. W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Languages*, pp. 1-18. March 1993.

[50] J. Stewart. *Reverse Engineering Hostile Code*. Security Focus Online. October 23, 2002. http://www.securityfocus.com/infocus/1637

[51] J. Stewart. *Alien Autopsy: Reverse Engineering Win32 Trojans on Linux.* Security Focus Online. November 14, 2002. http://www.securityfocus.com/infocus/1641

[52] The Executable and Linkable Format (ELF) File Specification. http://www.skyfree.org/linux/references/ELF_Format.pdf.

[53] H. Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proc. of the 7th Conference on Real-Time Computing Systems and Applications.* December 2000.

[54] C. Van Dyke. *An In-Depth Analysis of Common Software Vulnerabilities and Their Solutions.* M.S. Thesis, School of Electrical Engineering and Computer Science, Oregon State University. June 2004.

[55] C. Van Dyke and Ç. K. Koç. The Impact of Structural Obfuscation on Disassembly. Submitted to *ESORICS05, The European Symposium on Research in Computer Security.*

[56] C. Van Dyke and Ç. K. Koç. Realizing Structural Obfuscation. Submitted to *CCS05, The 12th ACM Conference on Computer and Communications Security.*

[57] C. Wang, J. Hill, J. Knight and J. Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report CS-2000. December 2000.

[58] C. Wang, J. Davidson, J. Hill and J. Knight. Protection of Software-based Survivability Mechanisms. In *Proc. of International Conference of Dependable Systems and Networks.* July 2001.

[59] G. Wroblewski. *General Method of Program Code Obfuscation.* Ph.D. Thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics. 2002.