A BIOINFORMATICS APPROACH TO THE SECURITY ANALYSIS OF BINARY EXECUTABLES

by

Scott Andrew Miller

Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Science in Computer Science with Information Technology Option

New Mexico Institute of Mining and Technology

Socorro, New Mexico

May, 2006

To those who picked me up when I fell, to those who dared me when I dreamed, and to those who allowed me to take great chances.

Scott Andrew Miller
New Mexico Institute of Mining and Technology
May, 2006

ABSTRACT

(holder)

ACKNOWLEDGMENTS

This report was typeset with LaTeX[1] by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for the New Mexico Institute of Mining and Technology report format was adapted from Gerald Arnold's modification of the LaTeX macro package for The University of Texas at Austin by Khe-Sing The.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

This report is accepted on behalf of the faculty of the Institute by the following committee:

Lorie M. Liebrock, Advisor

Scott Andrew Miller

Date

CHAPTER 1

INTRODUCTION

The goal of this thesis is to combine a variety of domains– Information Technology, Computer Science, Biology, and Mathematics – together into a coherent technique. To provide the necessary vocabulary, background, and analysis, the thesis must draw from each of the fields.

After this short introduction, a Background of the biological origins of this technique, the motivation for its cross-development, and the context of its application within the corpus related Computer-Science and Information Technology is presented. The development of Methods is considered next, a engineering problem that must first provide an adequate data representation and then provide a suitable matching method, the maximal segment pair measure. With the rigorous development of the methods, Results presents application of the developed method to several analysis problems. The effectiveness of the method is critically analyzed in the Discussion and a summary and potential areas for future work are presented in the Conclusion.

CHAPTER 2

BACKGROUND

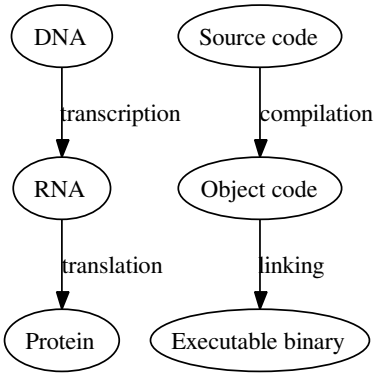
This thesis attempts to bridge developments in Biology analysis to an analytical need in Computer Science and Information Technology. It is first necessary to establish the analog from Biology that makes this possible, state the motivations for choosing such a cross-development, and finally present the Computer Science/ Information Technology context in which this cross-development occurs.

2.1 Biology Analog

Unlike computer processes, biological processes are manifest physically, can be observed directly, and are bound by physical properties. For example, biological function is dominated by proteins that are created in a series of translating steps[1]; a segment of DeoxyriboNucleic Acid (DNA) is selected for transcription into a transfer RiboNucleic Acid (tRNA), passed on to a ribosome that translates this into an Amino Acid, and those resulting Amino Acids are folded into proteins. In the forward direction, the process seems generally predictable. However, given an arbitrary protein, the identification

[1]In the Biology domain, "translation" has a very specific meaning as it is the process converting tRNA into amino acid chains. Throughout this document, "translation" should be considered in the general, non-Biology specific sense.

Figure 2.1: An analogy between the relationship between DNA and proteins next to the relationship between source code and binary executables.



of the responsible DNA segment is a non-trivial task.

Since 1985[2], the now-termed "bioinformatics" community has had relatively fast, heuristic methods to perform such searches. These searches have typically been to discover the most likely segment of DNA that codes for an arbitrary protein. The heuristic solutions are bounded by dynamic programming methods to determine the smallest number of mutations necessary to align and match a search string to its pair. These provably-minimal methods quickly become computationally infeasible[3]. Nevertheless, use of heuristic measures to estimate optimal solutions have proven extremely useful; there are now numerous multiple sequence alignment/search tools each with individual strengths and weaknesses[4]. These tools have been used to provide phylogenetic analysis[2] to determine evolutionary relationships, identification of conserved protein motifs to identify the structure and function of groups of proteins.

[2]Phylogenetic analysis considers the overall expression of a group of genes, the end result of complicated gene interactions.

Unfortunately, such methods do not exist for the analysis of computer binaries. Functionally, there is a translation sequence that parallels the transcription of DNA into protein: abstract source code is compiled into lower-level codes which ultimately are translated into machine-specific code. At each step, much as with the biological analog of Figure 2.1, pieces of the source code are removed by optimization or obscured by expansion. The end product, binary executable code, is much like a protein: given a corpus of source code, reverse engineering the source code that generated a given machine sequence is non-trivial[3].

2.2 Motivation

The need for direct computer binary analysis is exacerbated by interest in creating code that cannot be reverse-engineered nor detected. Much effort has been placed in what are deemed "code obfuscation" and "binary packing" techniques to obscure the nature of executable binary files[5, 6, 7, 8]. Some of these techniques have been specifically countered [9] or can be detected[**?**], but the only general solution is to capture (or more frequently to simulate) its execution[4]. To create a tool that is most effective against obfuscation and packing, the tool should make use of the direct results of such capture: either the effective execution stream or memory side-effects.

The work for this thesis is the novel application of a bioinformatics-

[3]For this analog, the term phylogeny may be expanded to describe the source code's expression as an executable. Thus, the overall computer behavior resulting from code execution is its phylogeny for some arbitrary source code.

[4]By previous analog, to capture/to simulate its phylogeny

inspired search algorithm, BLAST[1], to the problem of functionally analyzing binary executables. It should be noted that BLAST is not the newest development in bioinformatics nor is it the most complete model. However, in spite the simplicity of the underlying model, BLAST served as the gateway and benchmark for many other models and tools and remains of great value. The author hopes that the technique of this thesis will provide a sufficient analog to leverage the subsequent development of BLAST and similar bioinformatics techniques:

- Historical classification of never-before-seen code that can provide useful indications of the code's origins– compiler, author, and likely ancestors
- Identification of functional motifs already identified by expensive, conventional reverse-engineering techniques
- Functional prediction of unknown binaries

2.3 Related work

Although the need for tools for direct analysis increases, little emphasis has been placed on the development of such tools[10]. There is, however, much related work. In the context of this thesis, this work can be broadly grouped into three categories that are increasingly distant from the source code and increasingly close to the machine-instruction level:

1. Textual-based analysis of source code
2. Structural analysis of program control flow

3. Functional analysis

2.3.1 Source Code

At the highest level, analysis can focus on differences among versions of source code. This development has been of such importance to the development and management of large software projects that many of these developments are ubiquitous. Versioned source code repositories, code auditing tools, and code profiling tools are the most visible and prevalent of such tools. However, these tools require structured files[5] and have only limited use on the larger, more general class of unstructured files.

The consideration of unstructured files has recently been of increasing interest. As an example, Internet search engines must provide relevant results from unstructured, natural language. A particularly difficult search of natural language is that of plagiarism– finding two sections of natural language where one expresses the idea of the other without citing the other. A popular method for plagiarism detection has been to use overlap detection: reduce document into a number of smaller chunks, retain select chunks, compress those chunks into a digest, and then use the series of digests for analysis[11]. Overlap detection typically performs well with respect to false positives if an appropriate chunking length can be determined. Unfortunately, this optimal chunking length is difficult to determine as it is dynamic and related to dialect, subject, and author.

[5]Source code must necessarily be structured so that compilation is possible.

2.3.2 Structural Analysis

Identifying relevant sections of unstructured files, deemed chunks in the previous section, is simplified when considering the elements may provide structure. As one example, an analysis of system calls[12] has shown excellent resistance to code obfuscation techniques while providing viable identification and discrimination. The sequence and type of system calls is reduced to a signature that can be correlated using standard similarity measures to previously classified binaries. Unfortunately, these techniques do not necessarily aid in the determination of the nature and purpose and are furthermore reliant on binary executables making use of system calls[6].

A structural method to aid in this determination that is independent of operating system is that of control flow graph analysis. In this method, a binary is reduced into its basic blocks[7] and the interconnection graph of these blocks is used for analysis. An informal approach can use instrumentation inside basic blocks to provide run-time profiling[13]. This profiling is then used to reorganize the executable ideally into a canonical form. With a canonical form exists a structure that can be exploited by techniques already discussed.

If this analysis is further formalized, elements of mathematical graph theory can analyze and extract structural similarity. This formal modeling can be used to convert directly compiled binaries into some optimized, canonical form[14]. Alternatively, this modeling can be used to expose finer-grained

[6] As the malicious binary files become necessarily more complicated and recondite to survive, malicious binaries must either successfully mimic non-malicious binaries or make use of alternative channels.

[7] Sections of code that are unconditionally, sequentially executed

structural elements[10]. This particular development has attracted much interest in the security community as the approach has provided useful, viable tools necessary for the reverse engineering of increasingly complex malicious code.

2.3.3 Functional Analysis

For discussion in this paper, the finest grained consideration of a binary executable is deemed functional analysis. At this level, analysis is concerned with the individual sequences of instructions and their inter-correlations. The application of such analysis has been widely accepted in the anti-virus/anti-malware field; small signatures of the strings within binary executables can provide fast identification and isolation. Unfortunately, this type of signature-based detection is highly dependent on the arrival of new signatures and is fragile with respect to small code changes. If one could automatically determine the signatures[15], this system would not depend on external signature generation. Unfortunately, the automatic determination of signatures is no trivial matter.

As an alternative to looking for bit patterns, the bitstream can be disassembled into an instruction stream. This dissassembly provides not only an instruction stream but also insight as to memory side-effects. Analysis of the memory side effects is considered first.

In this analysis, the flow of data can be examined in terms of aliases, accesses to the same memory location[16]. Memory aliases are not always statically available, however, especially when a memory locations are dynamically computed. Nevertheless, such analysis still shows promising results and can

help to profile and to characterize binaries.

The other analysis, that of the instruction stream directly, has generally been in the domain of trained, experienced reverse engineers. Although there are some general techniques and approaches, automation of this process has typically suffered from either insufficient scope or unacceptable noise[**?**].

CHAPTER 3

METHODS

The search algorithm considered for cross-development is the Basic Local Alignment Search Tool (BLAST)[1]. As outlined in Section 2.2, this algorithm was chosen for its simplicity, superior memory alignment/usage characteristics, and broad-reaching applications to various emerging bioinformatics analysis. The algorithm is a heuristic optimization that attempts to find the optimal match and alignment, expressed by BLAST's "maximal segment pair" measure.

The development of this algorithm was heavily stepped in a carefully constructed first-order Markov Chain simulation to provide statistically significant results[25]. Because of wide use and acceptance, this analysis has been deemed "Karlin-Altschul" analysis after its two creators. Before this analysis can be used, the problem-at-hand must be first be appropriately represented and then the matching technique, an estimation of the maximal segment pair measure, can be developed.

3.1 Problem Representation

3.1.1 Concept

Biological systems have a number of well understood identities that make a partial match possible[1]. Matching binary executables poses many problems, especially on computing platforms that have variable-length machine instruction words[2].

To provide a basis to match the instructions in a binary executable, the instruction stream must be recovered either through disassembly or a capture of an instruction execution trace. When the assumption that the operands are most volatile[3] and that the operator class is the least volatile[4] are made, the match strength hierarchy of Table 3.1 can be constructed.

3.1.2 Format considerations

The most popular Linux distribution during the writing of this thesis was Ubuntu 5.10[18]. The installation CD for Ubuntu 5.10 includes some 95,100,406 machine instructions[5]. If this method is to efficiently and effectively

[1]For example, each DNA codon triplet (64 combinations) represents a single amino acid (20 amino acids). The amino acid Leucine can be represented by the DNA triplets CTT, CTC, CTA, or CTG. There is also a probability of amino acids being substituted for each other without changing the resulting protein, a phenomenon that can be modeled statistically[17].

[2]The Intel IA32 instruction set can have instruction lengths ranging from one byte to over fifteen bytes.

[3]Because of the complexity of memory aliases, simply moving data from memory into a machine register alters the operands without altering the function of anything that references this memory alias.

[4]As an example, there are various addition operators for signed, unsigned, floating point but these have the similar function– addition.

[5]The number of machine instructions in this distribution was obtained using the scaniso.py utility of ....

Match	Operator classes match	Operators match	Operands match
best	X	X	X
good	X	X	
fair	X		
none			

Table 3.1: A heuristic for scoring the strength of a match between two arbitrary instructions. This assumes that the operands are most likely to change and the operator class is least likely to change.

scan even a single distribution, the method must have compact data storage. Variable-Length Instruction Word streams such as those present in IA32 binaries create additional problems as they must be interpreted on every access. If this method is to remain flexible with respect to various instruction sets, it cannot rely upon fixed instruction lookup tables. To address these problems, a constant-length data representation based on the hashed values of disassembly output is considered.

Total length

The total storage length of an instruction in a constant-length representation is a compromise between clarity and total size. To provide the greatest clarity, every instruction is stored in its entirety, shorter instructions will have unused space in their storage. Shorter length representations require long instructions to be either decomposed into shorter instructions or truncated. Although it is possible to reduce longer instructions into a series of shorter instructions[6], such decomposition was not employed during for this

[6]The Pentium-4 architecture internally reduces the IA32 input stream into a number of "micro-ops", a Reduced Instruction Set Computer-based instruction stream.

thesis. Instead, this thesis made use of readily-available disassemblers.

The clarity/length compromise is not infinitely variable; most computing and storage architectures work efficiently on byte-boundaries. Furthermore, the cache systems available for most memory architectures are optimized for certain byte-based boundaries: byte, word, double-word, and quad-word. An efficient data storage format ought to reduce an instruction into its class, operator, and operand parts upon such boundaries.

Class determination

Ideally, the class of an operand would be based solely and referenced uniquely by its function. As an example, integer instructions, floating point instructions, data control instructions, and memory instructions might be used to decompose a system. One might use a finer-grained classification, reducing integer instructions into multiply, add, subtract, and divide classes. An alternative to this would be to group all multiplication, addition, subtraction, and division into separate categories regardless of their type (integer, floating point, etc.) Clearly, there are a number of ways to decompose logically an instruction set yet this creates a potential problem: each new instruction set encountered must be properly classified. Manual classification of architectures is a daunting task, especially for Complex Instruction Set Computers; the IA32 instruction set may contain over 1,000 machine instructions[7].

[7]Determining the exact number of machine instructions in the IA32 instruction set is difficult because the wide variety of extensions to the original Intel 8086 set. Intel produced additional instructions for the 8087, 286 and 386 platforms to utilize larger register sets and address spaces and has since extended this with the SSE, SSE2, SSE3, and MMX sets.

Because the operator and operands can be highly dependent, because not every machine operand may be used, and because instruction sets vary from machine to machine it is much more useful to consider 'species' of machine instructions generally reflected by mnemonics. Regardless of machine types, most machines posses 'ADD', 'MUL', 'JMP', etc. On this observation of a certain consistency of mnemonics is based a heuristic automatic classification scheme: the first n digits of an opcode can represent its class.

Class length

The previous consideration on class determination indicated that the first n digits of an opcode represent the class. If one assumes the broadest classification, the first digit of an opcode, and that opcodes are formed generally of letters, there are only 26 classes. In reality, not every letter is used and there are only 22 classes for IA32 systems[8] However, as the number of digits that generate the class increases, the number of potential classes increases as illustrated in Table 3.2.

As all but the broadest classification can be stored theoretically in a single byte and observations that IA32 classifications two digits or shorter can use a single byte, the question of clarity/length again arises. Although it may

AMD, a competitor to Intel, has also offered their own instruction set extension known as 3dnow!. IA32 has been extended by both Intel and AMD into IA64, extensions for 64-bit machines. To further complicate this determination is that literature generally considers the instruction mnemonics– 'ADD', 'MUL', 'MOV', etc. – which can each be assembled into different machine instructions depending on the type of their operands.

[8]To determine the number of IA32 classes, the header file 'opcode.h' for the GNU assembler was used as the operation list.

Class	1 digit	2 digits	3 digits
Theoretical	26/1 byte	676/2 bytes	17,576/3 bytes
Observed (IA32)	22/1 byte	137/1 byte	267/2 bytes

Table 3.2: The resulting number of classes/bytes required for direct storage when using the n-digit classification scheme. The observed IA32 statistics are based of the instruction list used by the GNU assembler and storage assumes an an ideal hashing function (no collisions).

be possible to develop specific hashing/truncation schemes for IA32 this must be developed for every new architecture to be considered. However, the use of some general hashing algorithm truncated to length will most likely cause two or more classes to have the same representation. As this is already a heuristic approach, it is assumed that collisions are not fatal to the procedure but it is desirable that these representation collisions be minimal.

Operator length

When considering the length required to adequately store the operator, it should be noted that both the class and operator fields can be used for uniquely identifying an operator. In addition to the considerations of hashing function/ truncation length for the operator field is that what are deemed 'duplications'– instructions whose class and operator fields are identical– are minimized. Observation of IA32 instructions, when removing operand-specific suffixes[9], indicates some 607 instruction species, something that can theoreti-

[9]For this, the operand suffixes 'b'-byte, 'w'-word, 'l'-long (double) word, 'q'-quadword, 's'-single, and 'p'-pop from stack were removed. There are additional extensions including 'd'-double. In general, presence of these extensions does not seem to be idiomatic and this filtering should be considered an estimation.

cally be stored in two bytes. From this observation, the sum of the class and operator lengths can be no less than two bytes.

Operands length

Because of the large possible variations in addressing modes, the variants possible based upon operand type, and the large number of registers present, operands can change greatly even when functionality remains the same. A consideration of operands is included to allow for consideration of 'perfect' matches, the operand length chosen to fill the total length.

Summary of format considerations

For convenience, a summary of the format considerations is presented in Table 3.3. This format expects and attempts to minimize various collisions in its representations and implicitly assumes that minimizing collisions will optimize the distribution of instructions by frequency. All of these assumptions need to be verified and characterized for this format to be useful to the method proposed by this thesis.

3.1.3 Format development

In order to verify the assumptions and create a design consistent with the format considerations of Table 3.3, a representative sample of binaries was used to evaluate and to develop the format.

Total length	– Fixed length – 4 or 8 bytes
Classification	– Based on the first n digits of opcode
Class length	– Truncation of hashing of class – Minimize collisions
Operator length	– Truncation of hash of entire operator – Minimize duplications (different instructions having the same class/operator values)
Operands length	– Hash of operands – Fill to total length

Table 3.3: A summary of the format considerations by field

Considerations

At the heart of the format is the use of a hashing function to reduce arbitrary length data into a fixed length digest and then truncating this digest. This process is used for three major reasons:

- Classification of any input is automatic and deterministic.
- Data is compressed into uniform segments
- Stored data cannot be used to reconstruct the intellectual property of the processed files[11]

The final point on protection of intellectual property is incredibly important to this tool with respect to security analysis. There is a subtle balance between providing enough information for security analysis and providing a generic tool to copy and to replicate technology. The introduction of collisions is necessarily a benefit; the many-to-one hashing relationship (especially within the operands) makes reconstruction of the instruction stream from the stored

data very difficult. Techniques such as using large pre-hashed tables[**?**] coupled with dynamic programming methods[19, 20] may be possible. The complexity of the underlying structures[10] and the short-lived nature of register information is believed to make this method less feasible and of higher cost than traditional reverse engineering efforts.

However, a static consideration of the use of the hashing/truncation involved is historically impossible. Two of the hash methods considered, MD5[21] and SHA1[22] were previously considered to be secure. However, both of these techniques have been compromised from a strict security standpoint[**?**]. This compromise does not reduce its usefulness for providing the data reduction under consideration. Their speed, small size, widespread implementation, and well-characterized weaknesses make them excellent candidates. A third, extremely simple hashing algorithm is also considered.

This simple hashing algorithm, arbitrarily called 'pow2', is considered for extremely succinct implementation and high speed. This hashing algorithm multiplies the previous digest by two and adds the current byte value. The algorithm is usually only of a pedagogic interest and is included for completeness.

Results

To choose a good hashing function and choose appropriate storage lengths, data was collected from the IA32-based installation sets for the top three Linux distributions at the time of development[18]: Ubuntu 5.10, Man-

[10]Particularly in the complexity of the alphabet, on the order of thousands of characters

drake 10.1, and SuSE 10.0. The instruction stream was recovered by static disassmbly using the GNU Binutils' Objdump[23]. This analysis attempted the disassembly of all files of all sections[11] and recorded using both Intel-type[12] assembly and AT&T-type assembly[13]. In short, the analysis was attempting to sample the worst, most diverse and potentially incorrect instruction stream present in the distribution. The summary of this experiment is shown in Table 3.4.

In general, there does not seem to be a justifiable benefit in using longer- length representations. Doubling the length from a single byte to two increases multiplies the potential number of bins by 2^8 yet measurement only indicates a doubling of the bins in use. Although increasing storage widths can completely eliminate duplications, this is at a cost of unused storage space that may be better utilized by the operand representation. Assuming that both the class and operand are stored in a single byte, the md5 representation that uses two digits for the class appears to have minimal duplications.

3.1.4 Format conclusions

By this reasoning and analysis, the instruction stream is reduced to a four-byte constant-length representation as in Table 3.5. The operator class is determined by the first two characters of the associated mnemonic. Each field is computed using the md5 algorithm and then truncated to one or two of its digest's top bytes. Although an operator cannot be uniquely identified by the

[11] `objdump -D`
[12] `objdump -M intel`
[13] `objdump -M att`

Class len	Hash	Grp bytes	Grp bins	Op bytes	Op bins	Duplications
2	pow2	1	83	1	227	130
	md5		103		243	48
	sha1		105		238	50
2	pow2	1	83	2	704	22
	md5		103		737	0
	sha1		105		741	0
2	pow2	2	83	1	227	130
	md5		134		243	44
	sha1		134		238	40
2	pow2	2	83	2	704	22
	md5		134		737	0
	sha1		134		741	0
Class len	Hash	Grp bytes	Grp bins	Op bytes	Op bins	Duplications
3	pow2	1	155	1	227	140
	md5		165		243	60
	sha1		166		238	64
3	pow2	1	155	2	704	46
	md5		165		737	2
	sha1		166		741	0
3	pow2	2	185	1	227	112
	md5		251		243	26
	sha1		252		238	28
3	pow2	2	185	2	704	46
	md5		251		737	2
	sha1		252		741	0
Class len	Hash	Grp bytes	Grp bins	Op bytes	Op bins	Duplications
4	pow2	1	197	1	227	410
	md5		209		243	326
	sha1		207		238	316
4	pow2	1	197	2	704	64
	md5		209		737	6
	sha1		207		741	0
4	pow2	2	383	1	227	86
	md5		423		243	12
	sha1		427		238	4
4	pow2	2	383	2	704	64
	md5		423		737	6
	sha1		427		741	0

Table 3.4: A summary of the format considerations for the worst-case instruction stream present in the top three Linux distribution at the time of research: Ubuntu 5.10, Mandrake 10.1, and SuSE 10.0. Duplications are defined in Subsection 3.1.2 and this table is used to evaluate the design criteria of Table 3.3

Byte 0	Byte 1	Byte 2	Byte 3
operator class	operator	operands	operands

Table 3.5: A four-byte, constant-length representation for instructions. A truncation of the md5 hash is used for each field and operator class is determined by the first two characters of the instruction's disassembled opcode.

operator field, it is relatively unique when considering both the operator class and operator fields together[14].

3.2 Estimating Maximal Segment Pair Measure

Using the representation format of the previous section, the concept of the BLAST maximal segment pair measure can be used to evaluate the relative match of sequences of instructions. The maximal segment pair measure is an integer score computed for an arbitrary alignment of two segments[15]. After aligning the two pairs with some offset, the match is sequentially extended in both directions. The score is incremented or decremented based upon the quality of the match included[16].

This score is maximized for the alignment: extending the matched region in either direction will not increase the score[24]. However, it is not computationally feasible nor necessary to compute the exact maximal segment pair; statistical relevance can be used to determine when extended a match is unlikely to increase its score and thus provide an approximate maximal segment pair. Such statistical relevance will be provided by Karlin-Altschul analysis[25], the

[14]There are only 48 collisions in the 504 unique operators observed.

[15]This process is similar to a discrete-time sequence correlation.

[16]For BLAST, the best (exact) match adds five to the score and the worst match subtracts four.

same technique used to provide statistical significance to the BLAST technique. Fundamentally, this analysis constructs a first-order Markov-chain model. In the method of this thesis as in BLAST, this analysis is used to choose the scoring constants, estimate when a match is statistically significant, and determine when score approximates the maximal segment pair measure.

The most difficult calculate and most critical part of the method of this thesis is the choice of the scoring constants. These scoring constants weight each of the match types of Table 3.1. The development of the an acceptable representation in the previous section made a great number of assumptions on what constituted a 'good' representation. Although many of these assumptions were validated and verified, the question remains 'Is this a viable technique?'

To answer this question with some confidence, a Karlin-Altschul analysis is used. For the consideration of this section, the top three Linux Distributions [17] at the time of development were represented as described in the previous section[18]. From this, the first two fields, the operand and operator, were used to construct a appropriate scoring constants.

3.2.1 Formal notation

A formal representation of the problem is required for the application of Karlin-Altschul analysis. Assume that the each unique instruction representation forms a single character. From this, a stream of instructions is

[17] Ubuntu 5.10, Mandrake 10.1, and SuSE 10.0[18]

[18] Only the sections of binary executables expected to have an instruction stream were disassembled into Intel-type assembly using GNU Binutil's objdump[23], `objdump -d -M intel`

formed from a unique alphabet $A = \{a_1, a_2, ..., a_r\}$. To form a random instruction stream, each of these letters can be sampled from A with probabilities $\{p_1, p_2, ..., p_r\}$. The scoring method outlined in Table 3.1 is formalized as an unambiguous scoring function for two alphabet letters i and j, function s.

$$s_{ij} = \begin{cases} x_1 | a_i = a_j \\ x_2 | a_i \neq a_j, operator(a_i) = operator(a_j), class(a_i) = class(a_j) \\ x_3 | a_i \neq a_j, operator(a_i) \neq operator(a_j), class(a_i) = class(a_j) \\ x_4 | a_i \neq a_j, operator(a_i) \neq operator(a_j), class(a_i) \neq class(a_j) \end{cases}$$

From this, an exact match scores x_1, operator matches score x_2, a class match scores x_3, and no match at all scores x_4. The results of this scoring function can be modeled by a random variable, S_{ij}, whose value is based upon the observed values of some instruction stream.

3.2.2 Model Development

The instruction stream of the Linux distributions observed include some 553,901,654 instructions that are reduced into an alphabet A of 2,199,784 letters. This reflects the great variety of the operands; when the operands are ignored, the same instruction stream can be expressed by an alphabet of only 466 letters. As the formation of a first-order Markov Chain model for Karlin-Altschul analysis requires the conditional probability of the next characters given any character. This requires every letter to be compared against every other letter ($O(n^2)$) and is thus computationally prohibitive for the full alphabet.

To limit the complexity, an assumption is made that exact matches are highly unlikely. From this, the selection of x_2, x_3, and x_4 can be evaluated and x_1 can be intelligently chosen. This selection is further bound by the

following constraints:

1. The scores must be strictly decreasing, $x_1 > x_2 > x_3 > x_4$, to ensure that the match semantics of Table 3.1 are observed.

2. The lowest score must be negative, $x_4 < 0$, to allow discrimination of matched and unmatched segments[25].

3. The highest score must be higher than the absolute value of the lowest score, $x_1 \geq -x_4$, to allow the potential for a positive score[25].

4. The expected score must negative, $E(S_{ij}) < 0$, to focus matching on smaller segments[25].

5. The maximum span between potential scores must be exactly one, $\delta = 1$, making every score from the minimum to maximum a possible score.

6. The weakest match must still have a non-negative value to provide intuitive relevance, $x_3 \geq 0$.

Developing S_{ij} is relatively straightforward as its probability is non-zero only at $\{x_1, x_2, x_3, x_4\}$, the value at each of these points representing the probability of each score. This model can be then be used to compute the Karlin-Altschul model and ultimately used to estimate a match scores required for statistical relevance.

3.2.3 Scoring Constant Selection

The resulting Karlin-Altschul model does not provide a strict fitness for relevant scores. Instead, it provides a model that provides the minimum

score for a match of length n to be relevant to some statistical significance. Throughout this constant selection, statistical significance is determined as the score for a sequence of length n below which are 99% of the scores for random sequences of length n from S_{ij}. It is likely that the maximal segment scores of short segments can never be statistically significant. To provide the finest-grained functional analysis possible, the length of the shortest statistically significant maximal segment, n_0, should be minimal.

As a secondary design consideration, it is desired that there be a large, usable variation in the quality of maximal segment pair scores. The minimal statistically significant score represents the measure's 'noise floor' and the theoretical score represents the measure's maximum signal. This is a convenient analog that allows for use of the standard signal-to-noise ratio and will be evaluated at the arbitrarily point at $n = 100$. A summary of this analysis is present in Table 3.6.

From this analysis, the shortest statistically significant maximal segment was four instructions long and there highest SNR was found in the family of $[x_2, x_4] = [5, -4]$. This warranted further analysis because adding the additional weighting factor for exact matches, x_1, might make the expected value non-negative[19], $E(S_{ij}) > 0$.

As presented in Table 3.7, the expected value remains strongly negative, implying that any of this family of scoring constants is viable. To provide a certain symmetry about the match strength and to leverage the

[19]Negative expected values focus matching on smaller strings

x_2	x_3	x_4	n_0	SNR (dB)
5	0	-4	4	23.968
5	1	-4	4	23.985
5	2	-4	4	24.008
5	3	-4	4	24.037
5	4	-4	4	24.076
9	1	-6	4	23.346
9	2	-6	4	23.360
9	4	-6	4	23.394
9	5	-6	4	23.415
9	7	-6	4	23.464

Table 3.6: Karlin-Altschul simulation results to provide scoring constants for statistical significance. This list is truncated from a listing of all possible scoring constants for $x_2 \leq 10$ to show results where the minimal relevant maximal segment length is at its lowest observed value, $n_0 = 4$. The highest Signal-to-Noise ratio at $n = 100$ is when $[x_2, x_3, x_4] = [5, 4, -4]$

x_3	SNR (dB)		$E(S_{ij})$	
0	23	968	-2	243.
1	23	985	-2	232.
2	24	008	-2	220.
3	24	037	-2	209.
4	24	076	-2	197.

Table 3.7: A closer inspection of the expected values for the scoring constant family $[x_2, x_4] = [5, -4]$. In each case, the minimum significant maximal segment was $n_0 = 4$ and the expected value remains strongly negative.

indicated strongest SNR for the choice of $\{x_2, x_3, x_4\}$, the match constants $[x_1, x_2, x_3, x_4] = [6, 5, 4, -4]$ will be used for the method of this thesis.

CHAPTER 4

RESULTS

CHAPTER 5

DISCUSSION

APPENDIX A

...

A.1 ...

REFERENCES

S. F. ALTSCHUL, W. GISH, W. MILLER, E. W. MYERS, and D. J. LIPMAN, "Basic local alignment search tool," *JOURNAL OF MOLECULAR BIOLOGY*, vol. 215, pp. 403 – 410, OCT 1990.

D. J. LIPMAN and W. R. PEARSON, "Rapid and sensitive protein similarity searches," *SCIENCE*, vol. 227, no. 4693, pp. 1435 – 1441, 1985.

L. L. Wang and T. T. Jiang, "On the complexity of multiple sequence alignment.," *Journal of computational biology*, vol. 1, no. 4, pp. 337 – 348, 1994.

C. C. Notredame, "Recent progress in multiple sequence alignment: a survey.," *Pharmacogenomics*, vol. 3, no. 1, pp. 131 – 144, 2002.

C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 290 – 299, 2003. Disassembly;Code obfuscation;Assembly codes;Software systems;.

N. Mehta and S. Clowes, "Shiva." ONLINE, 2003. http://www.securereality.com.au.

scut, "Burneye." ONLINE, 2002. http://packetstormsecurity.org/groups/teso/indexsize.html.

Z. Vrba, "cryptexec: Next-generation runtime binary encryption using on-demand function extraction." ONLINE, 2004. http://www.phrack.org/show.php?p=63&a=13.

C. Eagle, "Strike/counterstrike: Reverse engineering shiva." ONLINE, 2003. http://blackhat.com/presentations/bh-federal-03/bh-federal-03-eagle/bh-federal-03-eagle.pdf.

H. Flake, "Structural comparison of executable objects.," in *DIMVA*, pp. 161–173, 2004.

K. Monostori, R. Finkel, A. Zaslavsky, G. Hodasz, and M. Pataki, "Comparison of overlap detection techniques," *COMPUTATIONAL SCIENCE-ICCS 2002, PT I, PROCEEDINGS*, vol. 2329, pp. 51 – 60, 2002.

A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala, "Static analyzer of vicious executables (save)," in *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, (Washington, DC, USA), pp. 326–334, IEEE Computer Society, 2004.

J. R. LARUS and T. BALL, "Rewriting executable files to measure program behavior," *SOFTWARE-PRACTICE & EXPERIENCE*, vol. 24, pp. 197 – 218, FEB 1994.

M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pp. 169–186, USENIX Association, USENIX Association, Aug. 2003.

J. Z. Kolter and M. A. Maloof, "Learning to detect malicious executables in the wild," in *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, (New York, NY, USA), pp. 470–478, ACM Press, 2004.

S. K. Debray, R. Muth, and M. Weippert, "Alias analysis of executable code," in *Symposium on Principles of Programming Languages*, pp. 12–24, 1998.

M. O. Dayhoff, R. M. Schwartz, and B. Orcutt, "A model of evolutionary change in proteins," in *Atlas of Protein Sequence and Structure*, pp. 345 – 352, Washington, D.C.: National Biomedical Resource Foundation, 1978.

Distrowatch, "The top ten distrobutions." ONLINE, March 2006. http://distrowatch.com/dwres.php?resource=major.

V. A.J., "Error bounds for convolution codes and an asymptotically optimal decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, pp. 260–269, 1967.

L. R. RABINER, "A tutorial on hidden markov-models and selected applications in speech recognition," *PROCEEDINGS OF THE IEEE*, vol. 77, pp. 257 – 286, FEB 1989.

R. Rivest, "The MD5 Message-Digest Algorithm ." RFC 1321 (Informational), Apr. 1992.

D. Eastlake 3rd and P. Jones, "US Secure Hash Algorithm 1 (SHA1)." RFC 3174 (Informational), Sept. 2001.

GNU, "Gnu binutils." ONLINE, March 2006. http://www.gnu.org/software/binutils/.

P. H. SELLERS, "Pattern-recognition in genetic sequences by mismatch density," *BULLETIN OF MATHEMATICAL BIOLOGY*, vol. 46, no. 4, pp. 501 – 514, 1984.

S. KARLIN and S. F. ALTSCHUL, "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes," *PROCEEDINGS OF THE NATIONAL ACADEMY OF SCIENCES OF THE UNITED STATES OF AMERICA*, vol. 87, pp. 2264 – 2268, MAR 1990.