Hunting for Metamorphic JavaScript Malware

Mangesh Musale* Thomas H. Austin† Mark Stamp‡

Abstract

The Internet plays a major role in the propagation of malware. A recent trend is the infection of machines through web pages, often due to malicious code inserted in JavaScript. From the malware writer's perspective, one potential advantage of JavaScript is that powerful code obfuscation techniques can be applied to evade detection. In this research, we analyze metamorphic JavaScript malware. We compare the effectiveness of several static detection strategies and we quantify the degree of morphing required to defeat each of these techniques.

1 Introduction

Web browsers can be infected via malicious code inserted into JavaScript [29]. When a user visits such a compromised website, the malicious JavaScript is automatically loaded into the web browser. The majority of malicious JavaScript redirects the web browser to load malicious content from a remote server. This can be achieved through several means, such as adding an HTML `iframe` element to a page [23]. While there are relatively few ways to obfuscate HTML [38], JavaScript can be obfuscated, making it easier for malware writer to hide their true intent.

In this research, we analyze the proof-of-concept metamorphic JavaScript malware known as Transcriptase [35]. We consider a variety of static detection techniques and quantify the effectiveness of each. We then develop and analyze a strengthened version of Transcriptase, and we analyze the degree of morphing versus the difficulty of detection for each detection strategy considered.

This paper is organized as follows. In Section 2, we provide brief background information on malware, with an emphasis on metamorphism. Section 3 discusses Transcriptase, the metamorphic JavaScript that forms the basis for the research in this paper. Then in Section 4, we cover the Rhino JavaScript engine and discuss the modifications required for our experiments. Next, we briefly discuss the four metamorphic detection techniques that we analyze. Specifically, we consider a method based on Hidden Markov Models [37], an opcode graph similarity technique [26], a

*Department of Computer Science, San Jose State University
†Department of Computer Science, San Jose State University
‡Department of Computer Science, San Jose State University: stamp@cs.sjsu.edu

score inspired by simple substitution cryptanalysis [27], and Principle Component Analysis, based on the use of Singular Value Decomposition [17]. These topics are summarized in Section 5. Our experimental results for the Transcriptase metamorphic JavaScript appear in Section 6. We present our enhanced version of Transcriptase in Section 7 and provide experimental results for this more challenging case. Section 8 contains our conclusion and a consideration of possible future work.

2 Metamorphic Malware

Metamorphic malware changes its internal structure with each infection. The purpose is to avoid detection by obfuscating any signature (and, possibly, statistical properties as well), while retaining the essential functionality. Metamorphism can be viewed as a step in the progression of obfuscated malware, which we describe next.

One method to evade signature detection is to encrypt the malware. Even a weak encryption method can effectively hide a signature—each different key results in a different bit pattern. Packing (i.e., compressing) code can serve a similar purpose. From a virus writer's perspective, the weakness of encrypted code is that the decryption code is still subject to signature detection. Cascade, which appeared in 1987, was the first encrypted malware [24].

The next step in this evolution is polymorphic malware, where the body is encrypted, but the decryption code is morphed between generations. This method takes advantage of encryption to hide the malware signature, while also making detection of the decryption code more difficult. However, emulation or heuristics[1] can be used to detect polymorphics [5]. The virus known as 1260 (because it was only 1260 bytes in length), which appeared in 1990, is often cited as the first example of polymorphic malware [33].

Metamorphic malware is an advanced version of polymorphic malware, where the entire internal structure is morphed. Metamorphic malware is sometimes referred to as "body polymorphic." For well-designed metamorphic malware, encryption is not necessary, or even desirable. The virus known as Win95/Regswap, which appeared late in 1998, is generally credited as being the first example of metamorphic malware [33].

Next, we briefly discuss several elementary morphing strategies. In Section 5 we outline the advanced detection techniques that form the basis of the analysis in this paper.

2.1 Instruction Reordering

Independent instructions can be reordered without affecting program execution. If we have n independent instructions, we can generate $n!$ different morphed versions.

For example, consider the following code for the addition of two numbers:

[1]For example, self-decryption is not behavior that we typically expect to see in benign code.

```
1: int a = 10;
2: int b = 20;
3: int c = a + b;
```

Since statements 1 and 2, are independent, we can reorder the code as

```
2: int b = 20;
1: int a = 10;
3: int c = a + b;
```

Subroutine permutation is a particularly easy-to-implement version of instruction reordering. The Win32/Ghost is an example of a virus that uses subroutine permutation [40].

2.2 Instruction Substitution

Replacing an instruction or group of instructions by a functionally equivalent instruction or group of instructions is an effective metamorphic technique [36]. For example,

```
SUB EBX EBX
```

can be replaced by

```
AND EBX 0x0000
```

without any effect on program execution.

Register swapping is a weak form of instruction substitution. For example, `POP EAX` can be replaced with `POP EBX`, provided the `EBX` register is not in use. The previously mentioned Win95/Regswap virus relies entirely on register swapping for its morphing [32].

2.3 Garbage Code Insertion

Morphed code can be created by inserting instructions that are executed, but have no effect on the results [9]. Examples of such instructions include `NOP` and `ADD EBX 0` and we sequences of instructions, such as `INC` followed by `DEC` can also be employed Such "do nothing" instructions are the basis for the morphing in Win95/ZPerm [32].

In contrast to "do nothing code", there is "dead code," which is code that is never executed. Dead code is particularly useful for obfuscating statistical properties [20, 30] Automatic detection and removal of dead code can be made arbitrarily difficult by use of opaque predicates, for example [8].

3 Transcriptase

In biochemistry, a transcriptase is an enzyme that catalyzes the formation of ribonucleic acid (RNA) from a deoxyribonucleic acid (DNA) template during transcription [22]. The name Transcriptase was selected by the developer of the proof-of-concept metamorphic JavaScript malware in [35], which we consider in this paper.

The analogy between the biological process of transcriptase and this metamorphic Javascript generator is fairly tenuous.

In this section, we provide an overview of the Transcriptase metamorphic generator. A somewhat more detailed discussion can be found in [15].

Transcriptase infects all JavaScript files in the folder where it is executed. Each infection results in a morphed version of the malware being attached to the victim code. The purpose of the morphing is to evade signature-based detection.

The Transcriptase generator uses a custom meta-language to carry out its morphing. The meta-language code is compiled using a compiler written in JavaScript, and the compiler itself is part of the malware body. The advantage of defining a custom meta-language is that the malware writer can include information required to create highly morphed versions, without having the code grow uncontrollably over generations.

The format of instructions in Transcriptase is

```
(Identifier|Restrictions) meta-instruction
```

where `Identifier` and `Restrictions` are used for code obfuscation. Specifically, the `Identifier` is the unique identification of a statement in the script and `Restriction` is a list of statements that must be executed before the given statement. These meta-instructions are compiled to create actual instructions in the script. The overall morphing process consists of the following steps:

- Instruction Permutation
- Variable/Function Name Randomization
- Meta-language Symbol Resolution
- Code Creation
- Variable/Function Insertion

Next, we provide additional details on each of these steps.

3.1 Instruction Permutation

The Transcriptase compiler parses the meta-language code scope by scope, collecting all information about instructions and their corresponding restrictions. It analyzes the code in a given scope to determine an allowed permutation that does not violate any dependencies. For example, consider the code

```
(100|)var a=1;
(200|)var b=0;
(300|)def c=1;
(400|200)c+1(b);
(500|400,100, 200)c+n(b,a);
(600|200)xWScript.Echo(b);
```
(1)

Here, the first number is the identifier for the instruction, which is unique over the entire code and any numbers appearing after the "|" indicate dependencies. In this example, the assignment instruction 200 does not depend on any other instruction, while the instruction in 400 depends on the value of `b` which implies that it depends on instruction 200.

For the example in (1), the Transcriptase permutation function could produce the code

```
(300|)def c=1;
(100|)var a=1;
(200|)var b=0;
(400|200)c+1(b);
(500|400,100)c+n(b,a);
(600|500)xWScript.Echo(b);
```

Note that we have only changed the order of independent instructions in this example.

3.2 Variable and Function Name Randomization

The Transcriptase compiler searches for keywords such as `var`, `def`, and `function` and replaces the names with random strings. For example, given the code

```
function makeSquare (number){return number*number;}
var num = 1;                                                    (2)
def sqr = makeSquare(num);
```

the names are `makeSquare`, `number`, `num` and `sqr`. The compiler will generate random strings and map each string to one variable name and replace it within the current scope. After that, it will search for the same variable name in upper scopes until the global scope is reached. For the example in (2), the new form of code will be something like

```
function pkjuoledrbnx(qerdslds){return qerdslds*qerdslds;}
var hxedlkerd = 1;
def cadwkgd = pkjuoledrbnx(hxedlkerd);
```

3.3 Meta-Language Symbol Resolution

After permuting instructions and completing the name randomization, the Transcriptase compiler parses the meta-language code, translating meta-language symbols into valid JavaScript statements for statements. While deriving a JavaScript instruction, several meta-language symbols may be processed, each having a special meaning. The Transcriptase compiler interprets these symbols at runtime and generates appropriate javascript literals.

For example, consider the Transcriptase meta-language code

```
var number = #n1n#;
var str = #"Hello_World"#;
var exp = #xntruenx#;
```

Here, `#nyyyn#` is interpreted to mean that the value at position `yyy` is a numerical value, while `#"yyy"#` is interpreted to mean that the value at position `yyy` is a string value. Furthermore, `#xnyyynx#` is interpreted to mean that the value at `yyy` is a JavaScript literal, such as `true` or `false`. In all of these cases, `n` can be any integer. At this step, each symbol is replaced with a corresponding valid JavaScript language literal.

3.4 Variable and Function Insertion

During compilation, many variables and functions are defined. These are not yet inserted into code, but are instead collected in arrays. Only at the end of the code derivation are they actually inserted into the code. Consequently, additional dead code functions can be inserted between any two instructions in the global scope and dead code variables can be inserted between any two instructions in the current scope before they are used for the first time. This insertion process takes some time since the whole code has to be scanned several times to find potential insertion positions.

The random functions generated by Transcriptase perform arithmetic calculations and return integer values between 0 to 255. These values are then used to represent ASCII values of characters to derive strings.

4 Rhino

In 1997, Netscape began work on a project to develop a version of Netscape Navigator completely written in Java. As part of this effort, they developed a JavaScript engine, named the Rhino Project, which was also written in Java [25]. Rhino compiles JavaScript into Java classes and can operate in two modes—compiled or interpreted mode. In compiled mode, Rhino first compiles JavaScript and then converts it into Java bytecode. Then this bytecode can be run as a Java program. This process improves the execution time of JavaScript.

4.1 Architecture

The Rhino JavaScript engine consists of four basic building blocks, namely, parser, bytecode generator, interpreter, and the Just-in-Time (JIT) compiler. The JavaScript source code is first fed to the parser which converts the code into an Abstract Syntax Tree (AST). This AST is fed to the bytecode generator which produces the bytecode. The resulting bytecode is read by the Rhino interpreter, which converts it into machine code, with the help of the JIT. Finally, the machine code is executed [25]. A block diagram of the Rhino JavaScript engine is given in Figure 1.

More details on each of the components of the Rhino JavaScript engine can be found in [25] and the report [21]. Next, we briefly discuss modifications that we made to Rhino to facilitate the research presented in this paper.

4.2 Modification

We use Rhino as a tool to extract bytecodes from JavaScript. We then use the bytecodes in a variety of detection techniques. However, it was necessary to modify Rhino so that we could easily and efficiently extract the bytecodes.

As latency in compiling JavaScript directly affects the page load time, Rhino only compiles small JavaScript files. Also, there are different levels of optimization and we found that optimization can significantly transform the statistical properties which are important in our analyses. To deal with these issues, we modified Rhino so that it compiles JavaScript of any length, and we disabled optimization. And, since we are not interested in the class file, we directly extract bytecodes at compile time.

Figure 2 shows the output generated by unmodified Rhino after compilation. Note that after compiling JavaScript, we find a new class file generated in the same folder.

As with the unmodified Rhino compiler, our modified Rhino compiler also generates a class file. But, in addition to the class file, we also obtain a list of bytecodes. The list of bytecodes for the program compiled in Figure 2 appears in Figure 3. It is this list of opcodes that forms the basis for several of the scores that we consider later in this paper.

5 Detection Techniques

In this section, we summarize four statistical-based malware detection techniques , each of which has been previously analyzed in the context of metamorphic detection. In Sections 6 and 7, we apply these same techniques to the Transcriptase metamorphic JavaScript malware, and an enhanced version of Transcriptase, respectively.

5.1 Hidden Markov Models

A Hidden Markov Model (HMM) is a machine learning technique that was originally applied to the problem of metamorphic detection in [37]. In many subsequent studies, this HMM technique has been further analyzed, and it has often served as a baseline for comparison of proposed detection strategies [3, 4, 6, 12, 19, 26, 34].

In [11] and [20], the problem of defeating HMM-based detectors is considered. In [30], a fully functioning experimental metamorphic worm that can defeat HMM detection is developed and analyzed. We apply some of these same ideas to develop an enhanced version of Transcriptase that is more resistant to statistical detection techniques, such as HMMs.

A Markov chain consists of a series of state transitions, where the state transition probabilities are fixed. An HMM is a Markov chain where the state transitions are not directly observable. Instead, we have a series of observations that are statistically related to the "hidden" states via fixed probability distributions. We use the following

typical notation [31] to describe an HMM:

$$
\begin{array}{rcl}
T & = & \text{length of the observation sequence} \\
N & = & \text{number of states in the model} \\
M & = & \text{number of observation symbols} \\
Q & = & \{q_0, q_1, \ldots, q_{N-1}\} = \text{distinct states of the Markov process} \\
V & = & \{0, 1, \ldots, M-1\} = \text{set of possible observations} \\
A & = & \text{state transition probabilities} \\
B & = & \text{observation probability matrix} \\
\pi & = & \text{initial state distribution} \\
\mathcal{O} & = & (\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1}) = \text{observation sequence.}
\end{array}
$$

A generic view of an HMM is given in Figure 4, where the area above the dashed line is the hidden part. Note that the A matrix drives the (hidden) Markov process, while the B matrix relates the observations to the hidden states.

In an HMM, the matrices A, B, and π are row-stochastic, that is each row satisfies the conditions required of a discrete probability distribution. Together, these matrices define a model and we use the notation $\lambda = (A, B, \pi)$ to denote an HMM. The practical utility of HMMs derives largely from the fact that there are efficient algorithms for training (i.e., determining a suitable model $\lambda = (A, B, \pi)$ from a given series of observations) and scoring (i.e., determining $P(\mathcal{O} \mid \lambda)$).

In [37], HMMs are successfully applied to the metamorphic malware detection problem. An HMM is trained on opcode sequences extracted from a members of a given metamorphic family. The resulting model is then able to distinguish the metamorphic family from benign code with high accuracy.

Here, we use a similar approach to analyze the metamorphic JavaScript generator Transcriptase. As discussed in Section 4, we used our modified Rhino compiler to extract Java bytecodes from a number of members of the Transcriptase family An HMM was then trained on the resulting series of bytecodes, and the resulting HMM was used to score other members of the Transcriptase family and representative examples of benign JavaScript. Scores are computed as a log likelihood, and all scores are normalized to a per opcode basis. The resulting log likelihood per opcode (LLPO) score results are analyzed in detailed in Sections 6 and 7.

5.2 Opcode Graph Similarity

Anderson [1] describes a graph-based method for malware detection. In this method, an opcode sequence is extracted from a given malware and a weighted directed graph is constructed based on this sequence. Given a file to score, its opcode graph is also constructed and the graphs are compared. A simplified version of this score is considered in [26], where good results are obtained on several challenging metamorphic detection problems.

The same opcode graphs are used in [26] as in [1], but the scoring is simplified. Whereas Anderson [1] uses a relatively complex graph kernel analysis, in [26] the

graphs are directly compared using their adjacency matrices. Next, we discuss the process used to construct these opcode graphs, and the scoring technique used in [26].

The opcode graph is based on digraph frequencies in the extracted opcode sequence. A weighted directed graph is constructed, where nodes is added for each distinct opcode that appears. For each transition between opcode pairs, a directed edge is added, and edge weights are simply the transition probabilities.

For example, suppose we extract the opcode sequence in Table 1 from a given executable file. In this example, we obtain a sequence of 30 opcodes, with 8 distinct opcodes, namely,

`MOV`, `SUB`, `TEST`, `CALL`, `ADD`, `PUSH`, `AND`, and `LEA`.

Table 1: Opcode Sequence

Number	Opcode	Number	Opcode
1	`MOV`	16	`AND`
2	`SUB`	17	`PUSH`
3	`TEST`	18	`PUSH`
4	`CALL`	19	`MOV`
5	`SUB`	20	`CALL`
6	`ADD`	21	`CALL`
7	`PUSH`	22	`MOV`
8	`PUSH`	23	`SUB`
9	`AND`	24	`MOV`
10	`AND`	25	`SUB`
11	`AND`	26	`MOV`
12	`CALL`	27	`SUB`
13	`LEA`	28	`MOV`
14	`LEA`	29	`SUB`
15	`ADD`	30	`SUB`

The digraph distribution for the opcode sequence in Table 1 is given in Table 2. For this example, we have an opcode graph with 8 nodes, and for every non-zero entry in Table 2, we have a directed edge in the graph. For example, we have a directed edge from `TEST` to `CALL` and a directed edge from `AND` to `ADD`, but there is no directed edge from `CALL` to `TEST` or from `MOV` to `MOV`.

In Table 3 we have normalized the raw counts in Table 2. These numbers are used as the edge weights on the opcode graph. For example, the weight on the edge connecting `MOV` to `SUB` is $5/6$, while the weight on the edge from `AND` to `ADD` is 1.

Suppose that $A = \{a_{ij}\}$ is the edge weight matrix for one executable file, while $B = \{b_{ij}\}$ is the edge weight matrix from another executable file. Then we compute the

Table 2: Digram Count Matrix

	MOV	SUB	TEST	CALL	ADD	PUSH	AND	LEA
MOV	0	5	0	1	0	0	0	0
SUB	3	1	1	0	1	0	0	0
TEST	0	0	0	1	0	0	0	0
CALL	1	1	0	1	0	0	0	1
ADD	0	0	0	1	1	1	0	0
PUSH	1	0	0	0	0	3	1	1
AND	0	0	0	0	1	0	0	0
LEA	0	0	0	0	1	0	0	1

Table 3: Digram Probability Matrix

	MOV	SUB	TEST	CALL	ADD	PUSH	AND	LEA
MOV	0	$\frac{5}{6}$	0	$\frac{1}{6}$	0	0	0	0
SUB	$\frac{3}{7}$	$\frac{1}{7}$	$\frac{1}{7}$	0	$\frac{1}{7}$	0	0	0
TEST	0	0	0	1	0	0	0	0
CALL	$\frac{1}{4}$	$\frac{1}{4}$	0	$\frac{1}{4}$	0	0	0	$\frac{1}{4}$
ADD	0	0	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0
PUSH	$\frac{1}{6}$	0	0	0	0	$\frac{3}{6}$	$\frac{1}{6}$	$\frac{1}{6}$
AND	0	0	0	0	1	0	0	0
LEA	0	0	0	0	$\frac{1}{2}$	0	0	$\frac{1}{2}$

opcode graph similarity score as [26]

$$\text{score} = \frac{1}{N^2}\left(\sum_{i,j=0}^{N-1} |a_{i,j} - b_{i,j}|\right)^2$$

where N is the number of unique opcodes (or bytecodes) under consideration.

To be consistent with the paper [26], throughout this paper, we refer to this scoring process as an opcode graph similarity score. However, in subsequent sections we deal with bytecodes, so it would be more precise to call it a bytecode similarity score.

5.3 Simple Substitution Distance

Jakobsen [16] proposed a fast general algorithm for ciphertext-only attack on a simple substitution cipher. In [27], Jakobsen's algorithm was successfully applied to the problem of metamorphic detection. Jakobsen's algorithm decrypts a given ciphertext

message, relative to the language statistics. In the process, we obtain a measures of the distance between the underlying language statistics and the statistics of the given ciphertext.

For the metamorphic detection problem, the digraph statistics for a given metamorphic family correspond to the language statistics Jakobsen's attack, while the given file that we want to score corresponds to the ciphertext. Although the file we are scoring is not encrypted, it is obfuscated, and the simple substitution can be viewed as a form of obfuscation.

Next, we briefly discuss Jakobsen's algorithm, where we assume the underlying language is English, where only the 26 upper-case letters are used. Then we explain how to apply the algorithm to the malware detection problem.

Since we assume the language is English, the simple substitution key can be viewed as permutation on the alphabet, which we denote as

$$K = (k_1, k_2, k_3, \ldots, k_{26})$$

with each k_i corresponding to a distinct letter. The overall process is a hill climb, were at each iteration, the putative key K is modified by swapping a pair of elements. If the score improves, we maintain the swap; if not we revert to the pre-swap key. The swapping schedule proceeds as

$$\begin{array}{lccccccc}
\text{round 1:} & k_1|k_2 & k_2|k_3 & k_3|k_4 & \ldots & k_{23}|k_{24} & k_{24}|k_{25} & k_{25}|k_{26} \\
\text{round 2:} & k_1|k_3 & k_2|k_4 & k_3|k_5 & \ldots & k_{23}|k_{25} & k_{24}|k_{26} & \\
\text{round 3:} & k_1|k_4 & k_2|k_5 & k_3|k_6 & \ldots & k_{23}|k_{26} & & \\
\quad\vdots & \quad\vdots & & & \iddots & & & \\
\text{round 24:} & k_1|k_{24} & k_2|k_{25} & k_3|k_{26} & & & & \\
\text{round 25:} & k_1|k_{25} & k_2|k_{26} & & & & & \\
\text{round 26:} & k_1|k_{26} & & & & & &
\end{array}$$

where "—" indicates "swap." Also, we start over from the beginning of the swapping schedule each time the score improves, and hence we are done when we complete the listed $\binom{26}{2}$ swaps listed without any improvement in the score.

Let $E = \{e_{ij}\}$ be the digraph statistic matrix for English. That is, e_{ij} is the probability of letter i being followed by letter j in a large representative sample of English text. For example, the probability of `QS` is 0, while `TH` is the most common digraph in English.

Similarly, let $D = \{d_{ij}\}$ be the digraph matrix for a putative decryption of the ciphertext under consideration. Then the score used in Jakobsen's algorithm is

$$\text{score}(K) = \sum_{i,j=1}^{26} |d_{ij} - e_{ij}|. \tag{3}$$

Using normalized probabilities in E and D, we obtain a score that is independent of the ciphertext length.

Jakobsen's algorithm is extremely efficient since swapping elements k_i and k_j in the putative key corresponds to swapping rows i and j in the D matrix [16]. Consequently, we do not need to perform multiple trial decryptions—instead, we simply swap the appropriate row and column of D and compute the score in (3). Ironically, the overall algorithm is generally faster for longer ciphertext messages, since the scores are better and hence fewer swaps are required in the hill climb [13]. This is in stark contrast to the naïve method, which requires a trial decryption for each putative key tested.

The modifications needed to apply Jakobsen's algorithm to the metamorphic detection problem are straightforward. We gather opcode digraph statistics from a large set of family viruses and use these to form the analog of the E matrix. Then given a file to score, we extract its opcode digraph statistics which yields the analog of the D matrix. To score the file, we apply Jakobsen's algorithm with the score (3) for the final iteration being the score for the file. The lower the score, the better the file matches the statistics of the malware family. The intuition is that the simple substitution decryption process enables us to easily "see through" certain types of obfuscations.

A detailed example and analytic results can be found in [27]. The results show some improvement over HMM scores for certain challenging metamorphic families.

5.4 Singular Value Decomposition

Principle Component Analysis (PCA) consists of using linear algebraic techniques to reveal structure. The following simple analogy might serve to give the general idea [28]. Suppose that to explore a town in the American west, we apply the following algorithm:

1. Drive down the longest street in town.
2. When we see another long street, drive down it.
3. Continue until we have a reasonable map of the town.

It is likely that after exploring a relatively small number of these main streets, we will have a good idea of the general layout of the town. In PCA, the principle components correspond (roughly) to these long streets, and the process we use to find the principle components is somewhat analogous (with a few significant restrictions) to the process of looking for the longest streets. Below, we refer back to this analogy when discussing the SVD process.

There are many methods for computing the principle components, but perhaps the most popular is the Singular Value Decomposition (SVD). Recent work has shown that PCA is effective for metamorphic detection [12], with SVD specifically analyzed in [17]. Although the mathematics is fairly involved, and the training phase somewhat costly, the real beauty of PCA is that scoring is generally extremely efficient and often highly effective.

A detailed treatment of PCA would take us too far afield, and hence we only provide a brief overview, with the focus on SVD and its application to metamorphic

detection; see the recent paper [17] for a thorough treatment and analysis.

Let $A_{m\times n}$ be a matrix where column i is a set of m measurements for experiment i. That is, we have m measurements for each of n experiments, and we collect the results in the form of the matrix A. Then, assuming the mean of each measurement type is 0,

$$C_{m\times m} = \frac{1}{n}AA^T$$

is the covariance matrix. The diagonal elements of C are the variances for each measurement type, while the off-diagonal elements are the covariances between measurement types. Generally, large variances are the most interesting, since they indicate the most informative "directions" in the data. Furthermore, in the ideal case, all covariances are 0, since nonzero covariance indicates redundancy—the larger the magnitude of the covariance, the greater the redundancy. Consequently, the ideal case is a diagonal covariance matrix C, with a few large elements on the diagonal.

Of course, we have no control over C, since it is derived from our experimental data. But, by diagonalizing C, we can reveal structure that is otherwise "hidden" in the data. While there are many techniques for diagonalizing matrices, the SVD approach has certain advantages.

Let Y be an $n \times m$ matrix. Under suitable assumptions on Y, the SVD decomposes Y as

$$Y = USV^T$$

where S is a diagonal matrix containing the (square roots of) eigenvalues, and U and V contain the left and right singular vectors, respectively. The left singular vectors are the eigenvectors of YY^T and the right singular vectors are the eigenvectors of Y^TY. Note that YY^T is an $n\times n$ matrix, while Y^TY is $m\times m$. The point here is that the SVD to determine the eigenvalues and eigenvectors of the covariance matrix C. Since we have diagonalized the matrix in the process, and the eigenvalues appear on the diagonal, the largest eigenvalues reveal the most interesting "directions" in the data. The eigenvectors corresponding to the large eigenvalues reveal the most interesting structure. In the analogy mentioned above, these eigenvectors correspond to the main roads that we explored.

The action of a 2-dimensional shear matrix and its decomposition using SVD is illustrated in Figure 5. Shear transformations have a wide variety of uses, such as converting a letter in standard font to its italic or slanted form. This is a particularly simple example, but it does show that the SVD has a fairly intuitive physically interpretation.

Given any vector of the appropriate dimension, we can project it onto the space defined by the dominant eigenvectors. By doing so, we obtain a score that measures how well the vector matches the dominant structure of the covariance matrix. In fact, this score computation is generally extremely efficient, only requiring a small number of dot product computations.

To use the SVD for metamorphic detection, we proceed as follows. First, we extract the `.text` sections from n members of a given metamorphic family. Let m

be the maximum number of bytes in any of these `.text` sections and pad all others with 0, so that all are of length m. This is the training set. Next, we summarize training and scoring.

Training:

1. Let $V_1, V_2, \ldots, V_n$ be the (padded) `.text` sections from the n family viruses in the training set.
2. Treating each byte as floating point number, form the $m \times n$ matrix
$$A = [V_1, V_2, \ldots, V_n]$$
that is, V_i is the i^{th} column of A.
3. Use the SVD to determine the eigenvalues and eigenvectors of the covariance matrix $C = \frac{1}{n} AA^T$.
4. Let $u_1, u_2, \ldots, u_\ell$ be the eigenvectors corresponding to the ℓ dominant eigenvalues of C.
5. Project each V_i in the training set onto the space spanned by the eigenvectors $e_1, e_2, \ldots, e_\ell$. That is, compute
$$\Omega_i = \begin{bmatrix} V_i \cdot u_1 \\ V_i \cdot u_2 \\ \vdots \\ V_i \cdot u_\ell \end{bmatrix}. \tag{4}$$
6. Define the scoring matrix
$$\Delta = [\Omega_1, \Omega_2, \ldots, \Omega_n]$$
that is, column i of Δ is the weight vector corresponding to element V_i in the training set.

Next, we show how to use the scoring matrix Δ to score a given file.

Scoring:

1. Let X be the `.text` section of a file to score, padded with 0 to length m, if necessary.
2. Project X onto the eigenspace defined by $u_1, u_2, \ldots, u_\ell$, that is, compute
$$W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_\ell \end{bmatrix} = \begin{bmatrix} X \cdot u_1 \\ X \cdot u_2 \\ \vdots \\ X \cdot u_\ell \end{bmatrix}. \tag{5}$$

3. The desired score is computed as

$$\text{score} = \min_i d(W, \Omega_i)$$

where $d(x, y)$ is the Euclidean distance between the vectors x and y.

Suppose that we score a vector from the training set, that is, $X = V_i$ for some i. Then we obtain an exact match with Ω_i, and hence the score is 0. Conversely, for a vector that differs greatly from the training set, we expect to obtain a large score.

Finally, we note that the explanation here glosses over some subtle points and omits a few important practical issues. For a more complete description of this training and scoring process, see [17].

6 Transcriptase Experiments

In this section, we first discuss our experimental setup. Then we present detection results for the Transcriptase metamorphic family using each of the scores covered in Section 5, namely, Hidden Markov Models, opcode graph similarity, simple substitution distance, and Principle Component Analysis using SVD.

We extract a bytecode sequence from each Transcriptase malware using our modified Rhino JavaScript engine. The modification we made to Rhino are discussed in Section 4.2.

We performed all experiments on a machine with the following configuration.

- Model — Lenovo ThinkPad T530
- Processor — Intel Core i7-3110M @2.80GHz
- RAM — 8.00GB
- Java Compiler — Java 6
- Operating System — Ubuntu 12.10 (64-bit)

Each instance of Transcriptase carries its own morphing engine and the malware is highly metamorphic. For all experiments in this section, we used the same set of 100 distinct copies of Transcriptase. For our representative samples of benign code, we collected files from a variety of open source JavaScript libraries. The specific sources and number of files from each are given in Table 4.

To quantify the success of each experiment, we generate a Receiver Operating Characteristic (ROC) curve and compute the area under the curve (AUC). An ROC curve is a graph of the true positive rate versus the false positive rate as the threshold varies through the range of scores [7]. The AUC can be interpreted as the probability that a randomly selected positive instance scores higher than a randomly selected negative instance. This measure has been used in many previous studies and thus allows for a direct comparison to the results presented here.

Table 4: Benign JavaScript Sources

Source	Files
Cassandra Project [2]	5
DataTables Library [10]	10
Flanagan's book [14]	5
jQuery [18]	5
YUI Library [39]	15
total	40

6.1 HMM Score

Preliminary tests indicated that the number of hidden states had little effect on our results. Therefore, in all HMM experiment reported in this paper, we use $N = 2$ hidden states. We generated 100 copies of Transcriptase and extracted the bytecode sequence from each. We used 5-fold cross validation, that is, the set of 100 samples was partitioned into 5 equal subsets. We then trained on the first 4 subsets and used the resulting model to score the remaining subset. Another 4 models were trained, with a different subset reserved for scoring each time. In this way, we obtain a score for each of the 100 files in the training set, but we never score an instance that was used for training a given model. This process enables use to maximize the number of scores computed, while also smoothing out any bias in the data.

For the HMM score, a typical scatterplot appears in Figure 6. In this case, we have clear separation between the Transcriptase scores and the benign scores.

In fact, for all test cases considered we had similar separation between the Transcriptase malware scores and the benign scores. That is, we could set a threshold such that no false positives or false negatives would occur. Such cases generate the ROC curve in Figure 7, which yields an AUC of 1.0. From the detection perspective, this is the best possible case.

6.2 Opcode Graph, Simple Substitution, and SVD Scores

As with the HMM score, for all of the remaining scores tested (opcode graph similarity, simple substitution distance, SVD score), we also obtained ideal separation, resulting in ROC curves yielding AUC of 1.0. For the sake of brevity, we omit the graphs; see [21] for additional score graphs.

6.3 Discussion

The results in this section show that we can easily distinguish the Transcriptase malware from benign JavaScript. The HMM, opcode graph, and simple substitu-

tion scores are statistical in nature. The results for these scores indicate that the distribution of bytecodes in the disassembled Transcriptase malware is significantly different from that in typical benign JavaScript. On the other hand, the SVD score is structural in nature. The results for the SVD score show that the structure of the Transcriptase infected files also differ significantly from benign JavaScript. Although Transcriptase is highly metamorphic, the resulting family, as a group, is simply too different from benign code, making it relatively easy to detect.

In the next section, we try to remedy this situation. That is, we want to build on the impressive morphing capabilities of Transcriptase, by including features that will make the resulting code stealthier. This enhanced Transcriptase code has much more in common with the benign code—and thus it should be far more difficult to distinguish—as compared to the original Transcriptase. We perform a series of tests for each of the scores considered in this chapter to determine the effectiveness of our new-and-improved, stealthier version of Transcriptase.

7 Enhanced Transcriptase

From the analysis in Section 6 it is clear that we can easily detect Transcriptase with any of the four scoring methods tested. However, previous work has shown that we can defeat statistical-based scoring methods by carefully inserting dead code [30]. In this section, we discuss a modified version of the Transcriptase generator that allows for the insertion of selected amounts of JavaScript dead code, with the dead code taken from benign JavaScript. The goal is to take advantage of the sophisticated morphing capabilities available in Transcriptase, while making the resulting morphed code highly resistant to statistical-based detection. We test the resulting enhanced malware using the same scores used in the previous section, namely, HMM, opcode graph similarity, simple substitution distance, and SVD scores.

Our enhancement to Transcriptase operates as follows. We have created a service that runs continuously on a central server. Whenever Transcriptase is executed, it creates a connection with this central server. A number n is passed to the service, which then extracts n JavaScript functions from benign code, and these are sent to the malware. The malware then inserts these functions (as dead code) into its body. By adding dead code, the functionality remains the same, but the statistical characteristics of the bytecode sequences will tend to merge with that of benign code. Previous work has shown that such an approach can defeat statistical detection techniques [20, 30]. Therefore, we expect that at some level of dead code insertion, detection rates for our enhanced Transcriptase malware will deteriorate significantly. We would like to quantify the amount of dead code insertion versus scoring success for each of the scores under consideration. By quantifying the degree of morphing required, we can determine the practicality of this morphing strategy, and we can measure the relative robustness of each score with respect to the morphing techniques we have employed.

Due to the large number of experiments considered in this section, we omit scatter-

plots and ROC curves—we only provide the AUC statistic, as discussed in Section 6. Additional related results and graphs can be found in the report [21].

7.1 Score Results

As in the previous section, all experiments here are based on 100 malware instances, with the same 40 files used in our previous experiments as our representative benign code; see Table 4.

For each score, we conducted a series of experiments where each malware sample has additional dead code inserted, in the form of JavaScript functions selected from the set of benign files. In Table 5 we give the AUC for each of 30 different experiments for each of the 4 scores under consideration, where we have used the following abbreviations.

- HMM — HMM-based score, as discussed in Section 5.1.
- OGS — Opcode graph similarity score from Section 5.2.
- SSD — Simple substitution distance score in Section 5.3.
- SVD — The SVD-based score discussed in Section 5.4.

The individual AUC scores in Table 5 are graphed in Figure 8. For ease of comparison, the AUC values are plotted on the same axis in Figure 9.

Note that some of the AUC values in Table 5 are significantly less than 0.5. In general, if a binary classifier yields an AUC of p, we can simply switch the classification criteria to obtain an AUC of $1 - p$. In our experiments, we add dead code from benign code. The goal of inserting such code is to cause the properties of the morphed malware to merge with those of the benign code, making the morphed malware more difficult to detect. However, since we are extracting dead code from multiple, unrelated sources, for sufficiently high morphing rates it is possible for a particular score to yield an AUC of less than 0.5. More precisely, the scores tend to merge (i.e., the AUC approaches 0.5) at which point additional dead code can move the morphed malware scores beyond those for the benign set. This process is illustrated in Figure 10. In this example, the malware scores have essentially merged with the benign scores by the point where 10k dead code functions are inserted. Morphing significantly beyond this level causes the typical malware score to be greater than that of the typical benign score. The bottom line is that AUC values of less than 0.5 should be ignored, since it would be counterproductive (from the malware writer's perspective) to morph beyond the point where the AUC is 0.5.

The results in Table 5 are given in terms of the number of dead code functions inserted. In Figure 11, we graph the approximate percentage of dead code inserted relative to the number of functions. On average, each deadcode function contains about 0.85% of the code of the original Transcriptase malware. Note that this number is an (approximate) average, since the dead code functions are selected at random when morphing.

Table 5: Results for Modified Transcriptase

Deadcode	AUC Statistic			
Functions	HMM	SSD	OGS	SVD
100	1	1	1	0.8975
200	1	1	1	0.8872
300	1	1	1	0.8905
400	1	1	1	0.8848
500	1	1	1	0.8784
600	1	1	1	0.8671
700	1	1	1	0.8490
800	1	1	1	0.8366
900	1	1	1	0.8305
1000	1	1	1	0.8314
1500	1	1	1	0.8112
2000	1	1	1	0.7769
3000	1	1	1	0.7085
4000	1	1	1	0.6712
5000	0.9986	1	1	0.600
6000	0.9952	1	1	0.5651
7000	0.9803	1	1	0.5465
8000	0.9692	1	1	0.5263
9000	0.9653	1	1	0.4979
10000	0.9629	1	1	0.4723
11000	0.9544	0.9613	1	0.4544
12000	0.9471	0.9240	1	0.4364
13000	0.9339	0.7880	1	0.4138
14000	0.9330	0.1947	0.9999	0.3926
15000	0.9276	0.1707	0.9997	0.3817
16000	0.9185	0.1183	0.9997	0.3764
17000	0.9083	0.2583	0.9995	0.3658
18000	0.8985	0.3367	0.9991	0.3568
19000	0.8856	0.2400	0.9989	0.3427
20000	0.8693	0.1583	0.9987	0.3276

7.2 Discussion

The results in Table 5 show that our morphing has the greatest effect on the SVD score—at relatively low morphing rates the score has significant numbers of misclassifications. The simple substitution score also fails badly, but only at rates of morphing well in excess of 100%. The HMM score is relatively robust, yielding reasonable de-

tection rates beyond 150% morphing rates. The opcode graph similarity score clearly yields the strongest results, with virtually no degradation at the highest morphing rates tested.

The results in this section are broadly consistent with previous work on metamorphic detection [17, 26, 27, 37]. However, the opcode graph similarity scores are stronger than we would have expected. In addition, the relatively large variation between the first three scores (HMM, simple substitution, opcode graph similarity) is somewhat surprising.

The statistical nature of the HMM causes it to slowly degrade as the overall statistics of the malware and benign code merge. In contrast, the simple substitution score reaches a tipping point where "decryption" simply fails as the statistics merge. We believe that the (initially) surprising result for the opcode graph similarity score is largely explained by the presence of a relatively small number of uncommon bytecodes (with respect to the benign code) that appear in Transcriptase. The digraph weighting scheme used in the opcode graph score does not account for overall frequencies—only the relative frequencies at a given node are used when computing edge weights. Hence, rare bytecodes can continue to have a large effect on the score even after the overall bytecode statistics of the malware have converged toward that of the benign code. Selective code substitution would likely be an effective means for Transcriptase to defeat this particular score.

8 Conclusion and Future Work

In this paper, we analyzed Transcriptase, a metamorphic JavaScript malware. We experimented with several scoring methods that have previously been studied in the context of metamorphic detection and found that all were able to successfully detect this malware.

We then modified Transcriptase to make it stealthier. Our modified version was able to defeat most of the scoring techniques considered, Specifically, the SSD and SVD based scores were easily defeated, whereas the HMM score deteriorated, although much more slowly (as a function of the morphing percentage). In contrast, the OGS score continued to perform well, even at very high morphing rates. This work shows that it is possible to produce metamorphic JavaScript that is difficult to detect based on the Transcriptase generator.

Future work could include further enhancements to Transcriptase. For example, it should not be too difficult to incorporate morphing strategies that can defeat the OGS score. As another example, techniques such as those in [8] could be used to make the obfuscated code difficult to detect and remove. At a higher level, Transcriptase uses a custom designed meta-language to carry JavaScript. In its current implementation, there is a one-to-one static mapping between meta-language symbols and JavaScript literals. This static mapping could be made dynamic, so that in each new infection, the meta-language code will also change. In a sense, this would convert the current Transcriptase metamorphic generator into a meta-metamorphic generator. Although

not trivial to implement, such a modification might significantly reduce the degree of morphing required to evade detection.

Future work can also include developing a browser plugin to detect metamorphic (and related) JavaScript malware. As JavaScript malware executes in browsers and the JavaScript engine is a part of the browser, the modified Rhino compiler described in this paper can be used to develop a browser plugin that can analyze JavaScript at page load time. In this way, the work presented in this paper could be extended to provide dynamic runtime protection against many types of potential malware infections via the browser.

References

[1] B. Anderson, et al, Graph-based malware detection using dynamic analysis, *Journal of Computer Virology*, 7(4):247–258, 2011

[2] Apache Cassandra Project, `http://cassandra.apache.org/`

[3] S. Attaluri, S. McGhee, and M. Stamp, Profile hidden Markov models and metamorphic virus detection, *Journal in Computer Virology*, 5(2):151–169, 2009

[4] T. H. Austin, et al, Exploring hidden Markov models for virus analysis: A semantic approach, Proceedings of 46th Hawaii International Conference on System Sciences, 2013

[5] J. Aycock, *Computer Viruses and Malware*, Springer, 2006

[6] D. Baysa, R. M. Low, and M. Stamp, Structural entropy and metamorphic malware, *Journal of Computer Virology and Hacking Techniques*, 9(4):179—192, 2013

[7] A. P. Bradley, The use of the area under the roc curve in the evaluation of machine learning algorithms, *Journal Pattern Recognition*, 30(7):1145–1159, 1997

[8] C. Collberg, C. Thomborson, and D. Low, Manufacturing cheap, resilient, and stealthy opaque constructs, In Symposium on Principles of Programming Languages, pp. 184–196, 1998

[9] E. Daoud and I. Jebril, Computer virus strategies and detection methods, *International Journal of Open Problems in Computer Science and Mathematics*, 1(2):29–36, 2008, `http://www.emis.de/journals/IJOPCM/files/IJOPCM(vol.1.2.3.S.08).pdf`

[10] DataTables Library, `https://github.com/DataTables/DataTables`

[11] P. Desai and M. Stamp, A highly metamorphic virus generator, *International Journal of Multimedia Intelligence and Security*, 1(4):402–427, 2010

[12] S. Deshpande, Y. Park, and M. Stamp, Eigenvalue analysis for metamorphic detection, *Journal of Computer Virology and Hacking Techniques*, 10(1):53–65, 2014

[13] A. Dhavare, R. M. Low, and M. Stamp, Efficient cryptanalysis of homophonic substitution ciphers, *Cryptologia*, 37(3):250–281, 2013

[14] D. Flanagan, *JavaScript: The Definitive Guide*, 6th edition, O'Reilly Media, 2011

[15] P. Ferrie, Read the transcript, *Virus Bulletin*, May 2013, `https://www.virusbtn.com/virusbulletin/archive/2013/05/vb201305-Transcript`

[16] T. Jakobsen, A fast method for the cryptanalysis of substitution ciphers, *Cryptologia*, 19:265–274, 1995

[17] R. K. Jidigam, T. H. Austin, and M. Stamp, Singular value decomposition and metamorphic detection, to appear in *Journal of Computer Virology and Hacking Techniques*

[18] Jquery Library, `http://jquery.com/`

[19] J. Lee, T, H. Austin, and M. Stamp, Compression-based analysis of metamorphic malware, submitted for publication.

[20] D. Lin and M. Stamp, Hunting for undetectable metamorphic viruses, *Journal in Computer Virology*, 7(3):201–214, 2011

[21] M. Musale, Hunting for metamorphic JavaScript malware, Master's Report, Department of Computer Science, San Jose State University, 2014

[22] Oxford Dictionaries, transcriptase, `http://www.oxforddictionaries.com/us/definition/american_english/transcriptase`

[23] N. Provos, et al, All your iFRAMEs point to us, Proceedings of USENIX Security '08, pp. 1–15, 2008

[24] B. Rad, M. Masrom, and S. Ibrahim, Camouflage in malware: From encryption to metamorphism, *International Journal of Computer Science and Network Security*, 12(8):74–83, 2012

[25] Rhino documentation, `https://developer.mozilla.org/en-US/docs/Rhino_documentation`

[26] N. Runwal, R. M. Low, and M. Stamp, Opcode graph similarity and metamorphic detection, *Journal in Computer Virology*, 8(1-2):37–52, 2012

[27] G. Shanmugam, R. M. Low, and M. Stamp, Simple substitution distance and metamorphic detection, *Journal of Computer Virology and Hacking Techniques*, 9(3):159–170, 2013

[28] J. Shlens, A tutorial on Principal Component Analysis, `http://www.cs.cmu.edu/~elaw/papers/pca.pdf`

[29] SOPHOS Securtiy Threat Report, 2013, `http://www.sophos.com/en-us/medialibrary/PDFs/other/sophossecuritythreatreport2013.pdf`

[30] S. M. Sridhara and M. Stamp, Metamorphic worm that carries its own morphing engine, *Journal of Computer Virology and Hacking Techniques*, 9(2):49–58, 2013

[31] M. Stamp, A revealing introduction to hidden Markov models, 2012, http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf

[32] P. Szor and P. Ferrie, Hunting for metamorphic, Symantec Security Response, http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf

[33] P. Szor, *The Art of Computer Virus Research and Defense*, *Addison-Wesley Professional*, 2005

[34] A. H. Toderici and M. Stamp, Chi-squared distance and metamorphic virus detection, *Journal of Computer Virology and Hacking Techniques*, 9(1):1–14, 2013

[35] Transcriptase, http://spth.virii.lu/Transcriptase.rar

[36] R. Walenstein, et al, The design space of metamorphic malware, Proceedings of the 2nd International Conference on Information Warfare, 2007

[37] W. Wong and M. Stamp, Hunting for metamorphic engines, *Journal in Computer Virology*, 2(3):211–229, 2006

[38] W. Xu, F. Zhang, and S. Zhu, The power of obfuscation techniques in malicious JavaScript code: A measurement study, 2010, http://www.cse.psu.edu/~szhu/papers/malware.pdf

[39] YUI Library, http://yuilibrary.com/

[40] P. V. Zbitskiy, Code mutation techniques by means of formal grammars and automatons, *Journal in Computer Virology*, 5(3):199–207, 2009

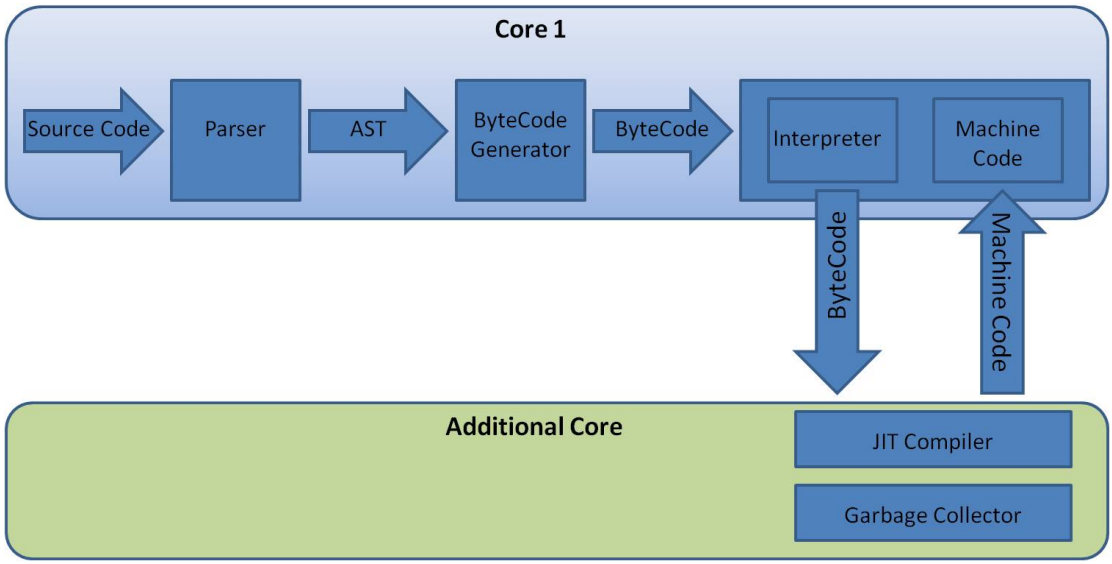


Figure 1: Rhino Block Diagram

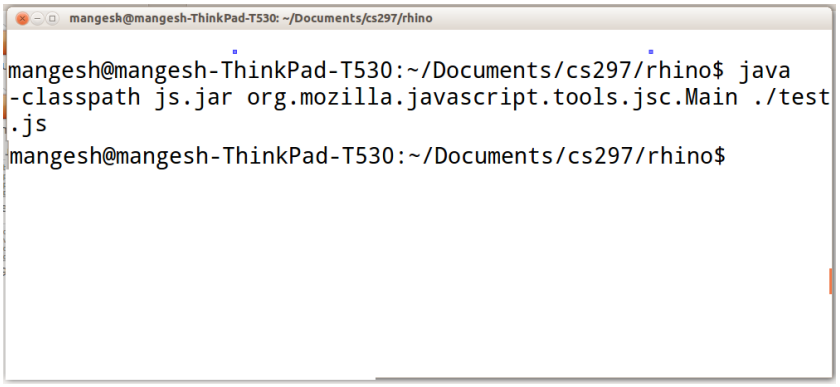


Figure 2: Sample Compilation Without Modification

```
mangesh@mangesh-ThinkPad-T530: ~/Documents/cs297/rhino

mangesh@mangesh-ThinkPad-T530:~/Documents/cs297/rhino$ java
-classpath js.jar org.mozilla.javascript.tools.jsc.Main ./test
.js
aload_0
invokespecial
aload_0
iconst_0
putfield
return
new
dup
invokespecial
```

Figure 3: Sample Compilation With Modification

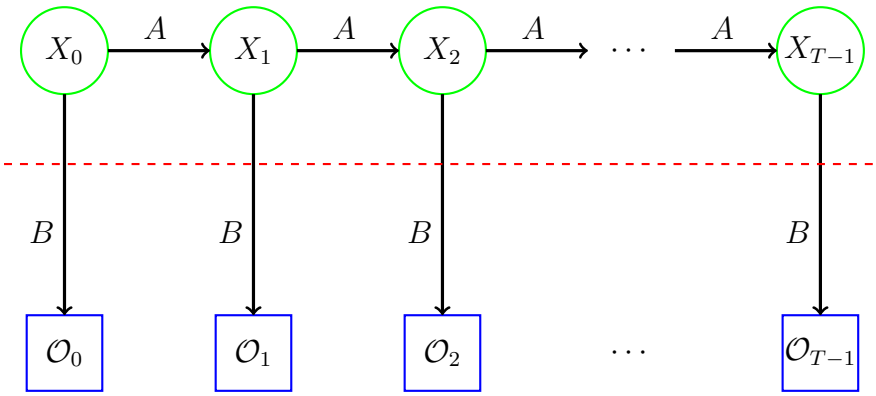


Figure 4: Hidden Markov Model

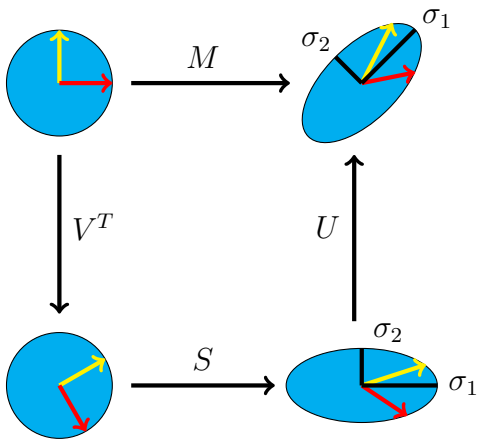


Figure 5: Matrix Transformation Using SVD

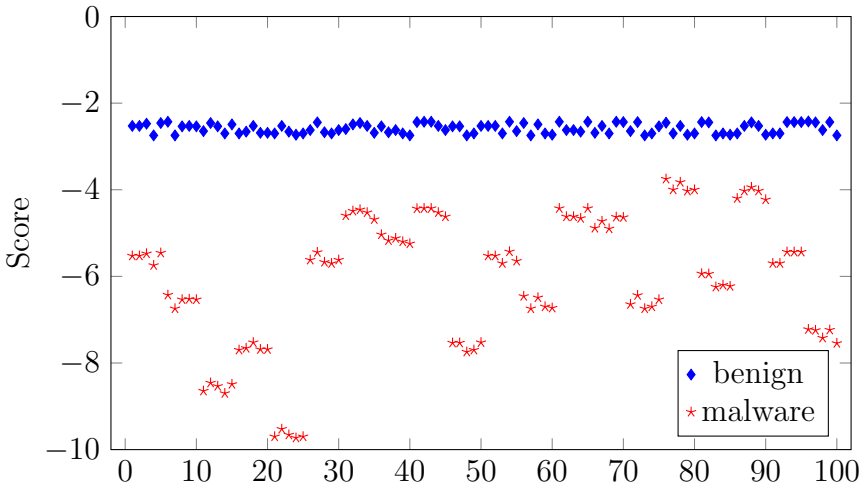


Figure 6: HMM Scores

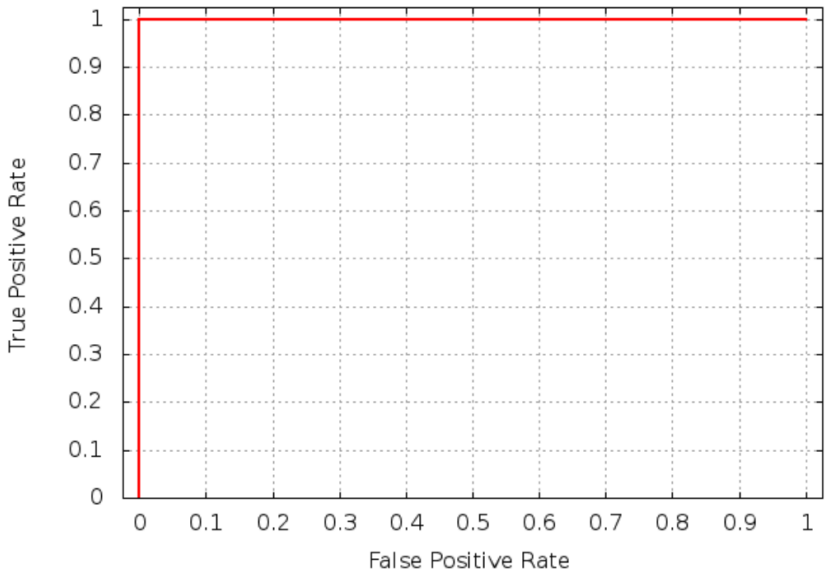


Figure 7: ROC Curve for HMM Score

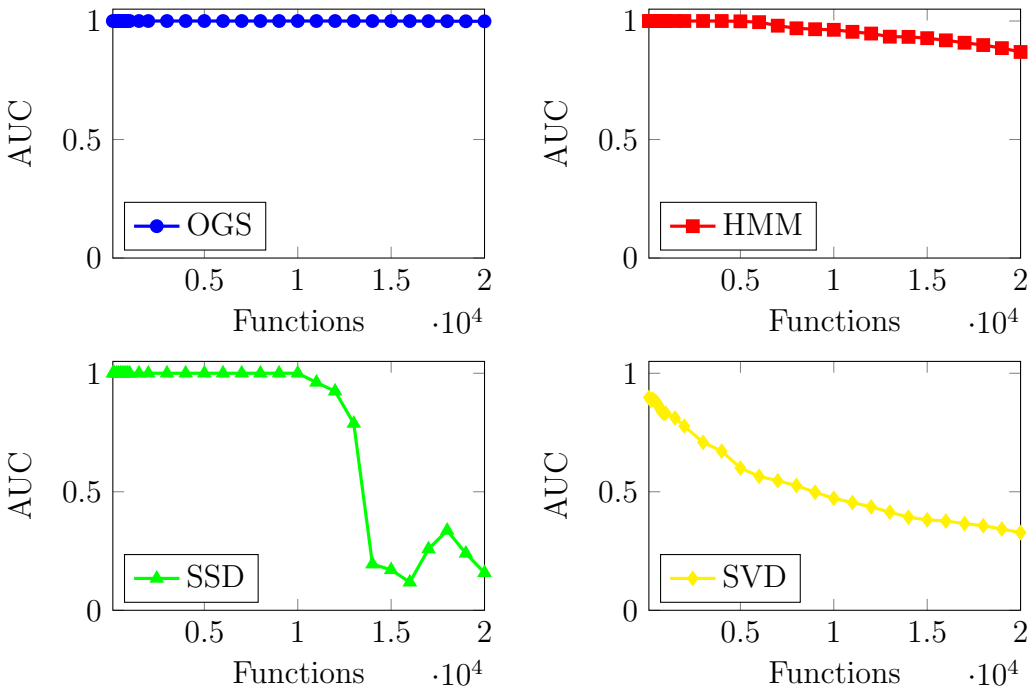


Figure 8: AUC Graphs

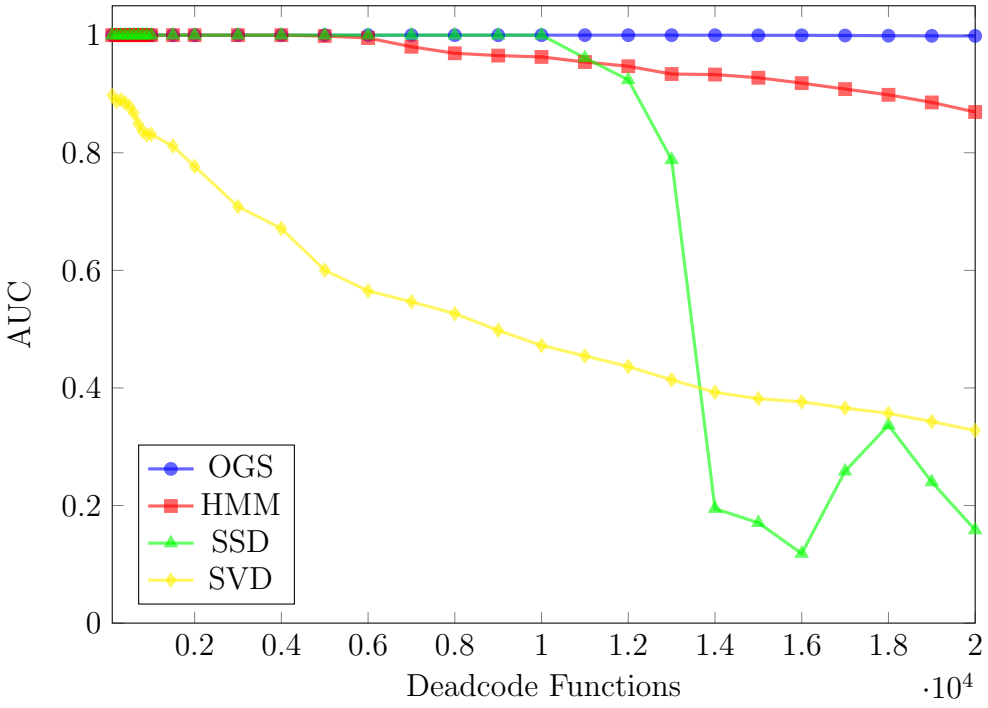
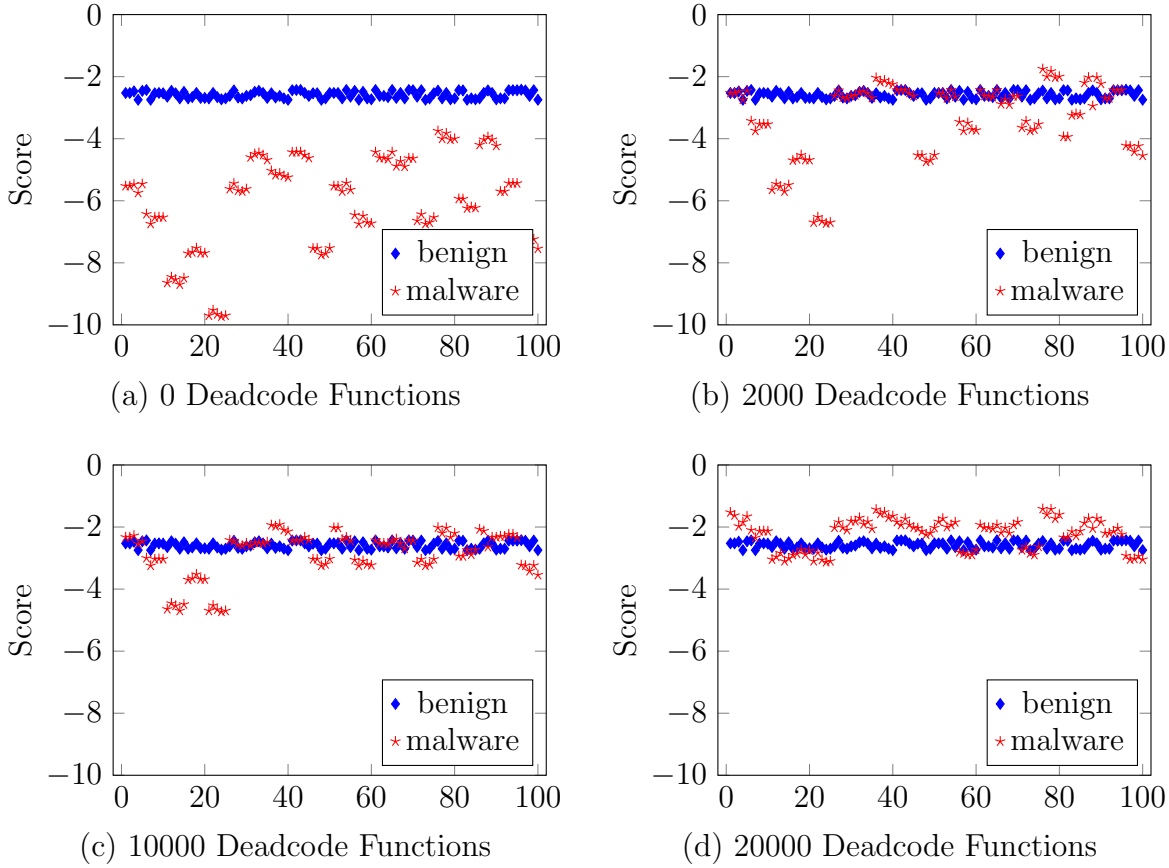


Figure 9: Combined AUC Graph



(a) 0 Deadcode Functions

(b) 2000 Deadcode Functions

(c) 10000 Deadcode Functions

(d) 20000 Deadcode Functions

Figure 10: Scatterplots as Morphing Rate Increases

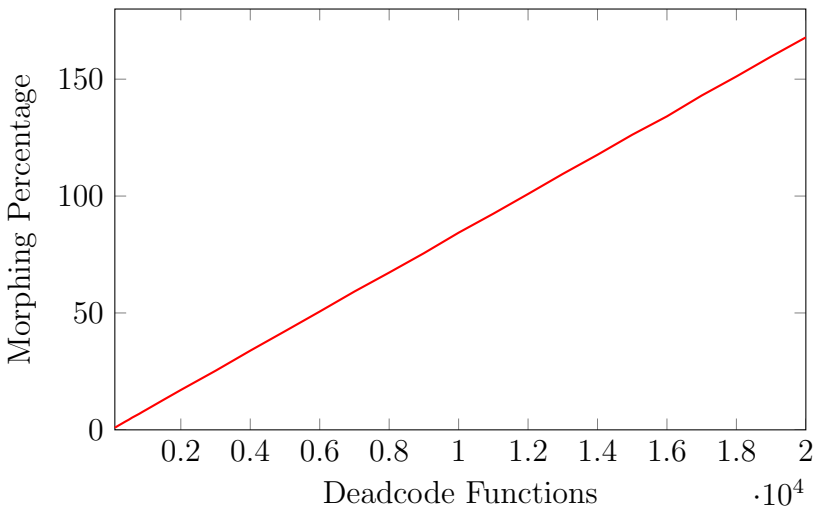


Figure 11: Morphing Percentage versus Number of Functions