ABSTRACT

Learning unknown sequences of instructions, given only observa-
tion is a complex problem. We propose that this can be solved
as a multi-way classification decision. Discussed are three dif-
f erent means by which support vector machines can be used to
handle multi-way classification, while selecting features which
will aid in the generalization of this problem. Our experiments
show marginal success using these methods, and we discuss the
various benefits and pitfalls of each classification approach.

1. INTRODUCTION

The ability to conceal the execution of a program instruction
stream is crucial to many security applications. Many secure
computing systems depend upon the privacy of instruction code
sequence to prevent adversaries from monitoring or reconstruc-
ting sensitive data. Further, trade-secret algorithms could not be
distributed publicly unless there was some guarantee that the
instruction sequence would be executed confidentially. Conse-
quently, “secure architectures” go to great lengths to obscure
and cryptographically protect the privacy of the execution of
sequences of instructions.

In this paper we present techniques by which machine learn-
ing algorithms can be used to attack these secure architectures
in an attempt to discern the identity of an unknown instruction
sequence. Specifically, we construct a classification problem and
use support vector machines to efficiently recognize hidden ma-
chine instructions given only the state of the computing system
before and after their execution.

1.1 Problem Scope and Model

In this type of problem, it is reasonable to assume that the
adversary has full observability of a computing system, run-
ned an application, before and after the hidden sequence of
instructions, but knows nothing during the actual execution of
these instructions, the “blackout period.” (The secure architec-
ture guarantees this privacy). Therefore, only the observable
state of the system before and after the blackout period can be
used as a source of data for our “attacking” machine learning
algorithms (our “classification problems”).

To this end, our machine learning algorithms cannot inter-
act with the system in an attempt to actively learn the hid-
 len sequence. The application could easily detect this tamper-
ing and terminate its operation permanently (using an online
third-party verification method). However, we do allow our
algorithms to perform a polynomial-time bounded amount of
“preprocessing” before it begins its observations. This would
allow for training which could improve results of instruction
sequences which might be more likely.

Finally, we assume that our algorithms have full knowledge of
all possible instructions which might be run during the blackout
period, as this information is nearly always documented.

1.2 Complexity Simplifications

Given an arbitrary computing system and application, this
problem is amazingly complex. Therefore we outline simplifica-
tions which make the problem manageable but can be general-
ized to handle more complex systems.

Since some architectures can have as many as 350 distinct in-
structions (Intel IA32), we restrict ourselves to allow only eight
possible instructions which can be executed within the black-
out period (Table 1). Eight instructions can be generalized to
n easily, simply by increasing our classification categorization.


<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mul C,A,B</td>
<td>Multiples C = A \cdot B (asymmetric)</td>
</tr>
<tr>
<td>div C,A,B</td>
<td>Divides C = A \div B (asymmetric)</td>
</tr>
<tr>
<td>slt C,A,R</td>
<td>Logical Left Shift C = A &lt;&lt; R, R = [0,31]</td>
</tr>
<tr>
<td>srl C,A,R</td>
<td>Logical Right Shift C = A &gt;&gt; R, R = [0,31]</td>
</tr>
<tr>
<td>and C,A,B</td>
<td>Logical And C = A \land B</td>
</tr>
<tr>
<td>or C,A,B</td>
<td>Logical Or C = A \lor B</td>
</tr>
<tr>
<td>add C,A,B</td>
<td>Addition C = A + B</td>
</tr>
<tr>
<td>sub C,A,B</td>
<td>Subtraction C = A - B (asymmetric)</td>
</tr>
</tbody>
</table>

Table 1: Possible Architectural Instructions

Furthermore, we define the “observability” of the system be-
fore and after the blackout period as two input values (operands)
and one output value (result). More complex views of a system
can be extrapolated from this model by simply duplicating the
classification problem for each possible input values, and prob-
abilistically determining what operands were used.

Finally, although numerous number representations are sup-
ported in different systems, we limit ourselves to unsigned inte-
gers in a standard 32-bit binary representation.

2. LEARNING MODELS

At first glance our problem of determining a sequence of in-
structions looks similar to function estimation, since this se-
quence merely calculates a function. However, a statistical re-
gression approach would not benefit us since deriving a match-
ing sequence of instruction from this would require extremely
high accuracy. Given that our operands can take any value be-
tween 0 and 2^{31} - 1, constructing an arbitrarily complex func-
tion approximation with near-zero error would be an astounding
feat.

A Hidden Markov Model which attempts to observe the in-
ternal state of the hidden instructions may also perform poorly
since, in our problem, we might not observe a sufficient num-er of different input/output pairs to discern a classification.
Further, the internal state changes of a long sequence of hidden
instructions produce no observable values.

Therefore we transform this problem to be a classification
decision, where different hidden instructions are grouped into
different classes. Since our model affords preprocessing time, we
train our classification algorithms to learn how many different
sequences of instructions would be categorized. This training is
used to estimate the hidden instruction sequence test samples.

It is worth noting, the classification of instructions into dif-
ferent classes is a probabilistic decision since differing instructions
can perform the same functional mapping, such as the example:
(slt C,A,R) = (mul C,A,B)

We present here three different multi-way classification tech-
niques which can be used for hidden instruction sequence clas-
classification, each with differing trade-offs. For all three methods, we use SVMs as the crucial binary classification algorithm.

## 2.1 Support Vector Machines

Since we wish to use a moderately large number of features, and we have prior knowledge of our classes, Support Vector Machines (SVMs) provide an excellent fit to our problem of supervised classification. SVMs [2] use a technique to change a feature dimensionality problem into a samples dimensionality problem, with the computational complexity focusing on inner kernel functions. The “C” parameter of an SVM adjusts the emphasis with which an SVM will minimize the weights (w) versus the margin violations (ξ), seen in the minimization of equation (1) constrained by equations (2).

\[ \frac{|w_1|}{2} + C \sum_{i=1}^{n} \xi_i \geq 1 - \xi_i \quad (\xi_i \geq 0) \]  

### 2.2 Output Codes

Output Codes (OCs) were introduced in [7] as a means to convert a multi-way classification problem into a sequence of binary classification problems. To classify using this method each particular class is assigned a binary vector of length \( m \) (\( +1 \) or \( -1 \)), or “codeword,” which corresponds to the outputs of \( m \) binary decision problems run on a single sample from that class. These codewords are unique to each class, so therefore we need at least \( \log(\ell) \) binary decisions to classify \( \ell \) categories.

For example, if we wanted to classify four categories \( C_1, C_2, C_3, C_4 \), using two decision problems \( d_1, d_2 \), we would define each codeword to be \( C_1 = \{d_1=-1, d_2=-1\} \), \( C_2 = \{d_1=-1, d_2=+1\} \), \( C_3 = \{d_1=+1, d_2=-1\} \), \( C_4 = \{d_1=+1, d_2=+1\} \). Both \( d_1 \) and \( d_2 \) will then be trained on samples from all four classes, labeled appropriately for each decision problem. When attempting to classify an unknown sample, both decision problems \( d_1 \) and \( d_2 \) will be run on the sample, and the resulting outputs are compared with the codewords of each class. In the case of sparse codewords, when an unknown sample produces outputs which do not match any of the classes, the class which has the closest Hamming distance is chosen.

We use SVMs to perform each binary classification, and determine our codewords according to a one-vs-all encoding. In this case \( \ell \) SVMs are trained for \( \ell \) classifications, where each SVM is a logical decision which predicts whether a sample is within a class or not within a class.

We choose this scheme since it assigns one SVM to specialize on each class, while still allowing for some errors. It also minimizes computational demands since only \( \ell \) SVMs must be trained. A one-vs-all approach makes sense when categorizing vastly different hidden instruction sequences since they may share very little structure in common.

### 2.3 Error-Correcting Output Codes

Error-Correcting Output Codes (ECOCs) are an extension to OCs which choose each class’s codeword so as to maximize the ability to correct errors in predicted codewords [3, 1]. This is accomplished by maximizing the Hamming distance between all pairs of codewords, or “columns.” Additionally, intra-codeword labelings (“rows”) should be uncorrelated to each other, therefore we maximize the Hamming distance between all rows of binary decision problem outputs.

We use SVMs to perform the binary classifications, and implement an exhaustive algorithm to determine optimal codewords. This algorithm maximizes both column and row constraints by requiring codewords to be of length \( 2^k - 1 \), and assigns labels in a tree-wise fashion.

We use this method to optimally classify sequences of instructions which are very similar in nature, despite ECOC’s high computational cost (an eight-way classification requires 127 SVMs to be trained). If we find situations where ECOCs perform well, there are other suboptimal codeword selection algorithms we can use which give similar results but lower the computational requirements.

## 2.4 Decision Trees

Decision trees are another means to classify a multi-way problem using a series of binary decisions [6]. A balanced binary decision tree which classifies \( N \) categories breaks the problem into \( D = \sum_{i=2}^{N} \frac{N}{i} \) distinct binary decision problems (when \( N = 2^k \) for some \( i \)). Each decision problem, or SVM-node, within the tree addresses a specific question to be answered.

To train a decision tree, each sample is labeled in \( D \) different ways, and trained on each SVM-node. To classify a new sample, the “root” decision is applied to the sample, determining if the sample belongs in the 1st \( \frac{N}{2} \) classes or the 2nd \( \frac{N}{2} \) classes. This is repeated iteratively down the tree. Figure 1 shows a binary decision tree for eight classes.

![Figure 1: 8-way classification using a decision tree](image)

When classifying hidden instruction sequences, we do know a lot about the interactions and likelihood of the different instructions a priori. We can therefore create fixed tree structures which represent prior knowledge such as instruction family or domain of application. There is no need to build a tree structure from training samples, as is done in problems which do not have such concrete anterior information.

We use decision trees to classify hidden instruction sequences we have familiar knowledge of because they offer an attractive tradeoff between the number of SVMs which must be trained, and the classification potential of each decision.

## 3. FEATURES

Feature selection is critical to any learning problem; if we were to select our final answer as a feature, no learning is necessary at all. Since we can only observe operand and result values, features must be constructed using only this limited information.

### 3.1 Ideal Features

Ideally, we could choose features which immediately solve our problem. To classify a mul or div, we would simply add a is-a-multiply feature which compares the result with the multiplication of the operands.

This approach does not generalize, however, when we consider sequences of instructions. In this case the number of feature would grow at a factorial rate. Even though SVMs handle a high number of features well, 3.6 million features to classify a short 10 instruction sequence is simply not tractable.

### 3.2 Vectorized Distance Metrics

To allow for better generalization, we introduce ten functions which determine differing notions of “distance” between bit-vectors:

- \( f_1 \): Hamming distance
- \( f_2 \): Position-independent Hamming distance (difference of 1's-count)
- \( f_3 \): Needleman-Wunsch distance [4] with \( G = 2 \)
- \( f_4 \): Longest consecutive matches (position-dependent)
- \( f_5 \): Longest consecutive matches (position-independent)
- \( f_6 \): Q-gram match count (\( Q = 2 \))
$f_7$: Levenstein distance modified to weight insertions according to bit positions

$f_8$: Directional Hamming distance, where $0 \rightarrow 1$ transitions are weighted increasingly according to bit position from MSB to LSB and $1 \rightarrow 0$ transitions are weighted decreasingly according to bit position from MSB to LSB

$f_9$: “Greater-Than” 32-bit unsigned integer representation, binary output

$f_{10}$: Diagonalizing “one’s-sum,” where bit $i$ of $2^{nd}$ vector is tested to be 1 or 0, if 1, sum all 1’s from $i$ to LSB in $1^{st}$ vector

For each of these “feature-functions” we generate two features: the first computes the distance between the first operand and the result, the second considers the second operand and result. This method captures the independent relationships between both operands and the result values.

We choose features $f_{1...2}$ to capture large, bitwise changes in values, features $f_{3...7}$ to detect arithmetic, large value changes, and features $f_{8...10}$ to detect unbalanced bit changes that only effect a small number of bits.

3.3 Feature Selection

To determine which features are appropriate for different decisions, we must analyze how well individual and multi-wise features perform on our problems.

Removing feature which do not aid in classification lowers the amount of noise we see, and therefore reduces misclassifications. It also helps reduce computation time. Decision trees and OCs benefit especially since each SVM-node, or one-vs-all decision can be tailored to benefit specific class categorizations. ECOCs, however, only benefit when a feature is removed entirely from all $2^k-1$ decisions. This is because intra-codeword labelings are maximally independent, so a single SVM decision cannot be tailored to a single classification type.

In the “SingleOp” decision tree classification problem, we attempt to classify a single hidden instruction into eight possible categories. For this we use a decision tree as in Figure 1, and classes: $C_{1...8} = \{mul, div, sll, srl, and, or, add, sub\}.$

Table 2 gives a breakdown of the effectiveness of each individual feature-functions on the seven SVM-node within the tree. We note error rates which are indicative of favorable features, therefore each decision $d_i$ should include only those features which are favorable in its corresponding row.

We also tested all multi-wise combinations of the feature-functions and found no case for a combination of two or more features changed our decision of which features to keep.

$$
\begin{array}{cccccccc}
\text{Test} & f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 & f_9 & f_{10} \\
\hline
d_1 & .20 & .31 & .25 & .29 & .23 & .27 & .28 & .33 & .47 & .44 \\
d_2 & .15 & .15 & .19 & .19 & .19 & .14 & .16 & .40 & .45 & .27 \\
d_3 & .01 & .09 & .32 & .09 & .19 & .02 & .51 & .11 & .50 & .48 \\
d_4 & .42 & .23 & .31 & .00 & .11 & .01 & .00 & .00 & .00 & .00 \\
d_5 & .20 & .52 & .50 & .50 & .43 & .50 & .50 & .29 & .29 & .31 \\
d_6 & .50 & .49 & .50 & .50 & .44 & .50 & .50 & .45 & .45 & .44 \\
d_7 & .07 & .38 & .38 & .50 & .25 & .48 & .25 & .48 & .05 & .26 \\
\end{array}
$$

Table 2: Error rate for individual feature-functions (chance=.50)

We performed such calculations for all the experiments described in Section 4.1, for OCs, ECOCs, and Decision Trees. While for decision trees and OCs, we can remove features for specific tests, we only removed features for ECOC experiments when a feature-function never improved our error rate. For example, in Table 2 $f_1$ completely dominates both $f_2$ and $f_6$ in performance. Therefore we remove features $f_2$ and $f_6$, reducing erroneous noise and computation time.

4. EXPERIMENTATION

We show here four experiments that demonstrate the benefits and pitfalls of our three multi-way classification techniques using the distance features discussed in Section 3.2.

We generate separate (but interchangeable) training and test samples using a configurable generation program we had written. We select operand values from a uniform distribution $U[0, 2^{32} - 1]$, create the user specified features, and use sample relabeling and feature removal scripts to dovetail each sample to each decision problem. We normalize all features to be within the range $[0, 1]$.

For each basic binary SVM decision we use the SvmFu tool [8], however our decision tree, OC, ECOC multi-way classification methods are again implemented by a number of bookkeeping programs and scripts. Both linear and Gaussian kernels are compared throughout the experimentation to expose issues of overfitting and performance costs. Experimentation found that the optimal Gaussian kernel parameter was $\sigma = 1.9$ which we used. Finally, we use a $C$ penalty factor of 100 for each SVM decision to apply a moderate regularization which we found to be optimal in practice.

4.1 Experiment Discussion

First, we show the “SingleOp” classification described in Section 3.3. This experiment tests how our algorithms perform on a simple classification problem, a problem which we know a lot about a priori. One benefit of this is that we can construct our decision tree in a manner which minimizes test error, as we can see in Table 3, the decision tree model outperform all others.

This also illustrate how different algorithms respond to relatively few training point, as we see in Figure 2. The decision tree algorithms clearly performs best, for both linear and Gaussian kernels. This is likely because it takes a very small number of samples to give each SVM-node a preference to $+1$ or $-1$ given a sample’s class.

4. EXPERIMENTATION
meaningful computations for applications, so we have a reasonable amount of knowledge of their traits.

In Figure 3 we see again that the decision tree algorithms perform better with less training samples, although in this case the ECOC method performs best in the long run. More blatant though, are the aberrations caused by different kernel selections. We see that error rates of linear kernels strictly dominate those of Gaussian kernels, demonstrating how optimal parameterization of a Gaussian kernel outperforms linear kernels.

Finally, “SeqSz” is a classification problem concerning eight instruction sequences with lengths varying from one to eight. We see here that ECOCs outperform decision trees since our classes are not well partitioned into binary decisions.

### 4.2 Biased Input Data Values

All of the preceding experiments used a uniform distribution of operand values, however this is unrealistic in practice. Experimentation indicates that most values are ≤ 128 in a real computing system, therefore containing a large number of zeros within their bit-vectors. We used the PIN Instrumentation Tool [5] to extract random value traces from common applications. Table 4 duplicates Table 3 for Gaussian kernels, but using biased real-world operand values.

We see that biased data skews the classification error rates of all algorithms. Since many low values appear as operands, this drastically reduces the amount of distinction between different classes. More formally, this bias is reducing the mutual information score of all features.

<table>
<thead>
<tr>
<th>Classification</th>
<th>SingleOps</th>
<th>OpInMxSz</th>
<th>DiffSeqs</th>
<th>SeqSz</th>
</tr>
</thead>
<tbody>
<tr>
<td>OC (Gaussian)</td>
<td>46.3%</td>
<td>55.0%</td>
<td>54.1%</td>
<td>47.7%</td>
</tr>
<tr>
<td>ECOC (Gaussian)</td>
<td>50.4%</td>
<td>52.0%</td>
<td>54.2%</td>
<td>77.6%</td>
</tr>
<tr>
<td>Tree (Gaussian)</td>
<td>45.6%</td>
<td>53.0%</td>
<td>53.6%</td>
<td>75.0%</td>
</tr>
</tbody>
</table>

Table 4: Error rate using biased values (chance=87.5%)

### 5. LEARNING ANALYSIS AND TRENDS

We see from Table 3 that a Gaussian kernel almost always performs better than a linear kernel, given our parameter σ = 1.9. However, small values of σ can lead to overfitting, and larger values of σ would not be able to classify much at all. We see this overfitting in Figure 4 (SingleOp test, using decision trees) as we decrease σ < 0.5 (the test error no longer correlates with the training error). We also see the increase in error rate as our sigma grows too large.

Regularization is also a crucial parameter which we considered when creating our model. Figure 5(a) shows the effect of modifying our C penalty factor from 0 to 150. A small C focuses on minimizing the SVM weights instead of the violations. This minimization corresponds to neither the test or training points, so both see high error rates. Large C values reduce training

![Figure 4: Training and test error rates with σ](Image)

error rates, but overfit to the data, increasing test error. Interestingly, due to the inherent structure of the SingleOp problem, the test error rate is 0.35 even with C=∞.

The number of Support Vectors (SV) used also decreases as C increases, as seen in Figure 5(b) (SV count for entire decision tree). This is because a lower C misclassifies more points, and all misclassified points become a SV with some violation ξ.

![Figure 5: (a) Error rates with C (b) SV counts with C](Image)

### 5.1 ECOC/Output Codes Characteristics

Both error-correcting output codes and one-vs-all output codes appeared to perform similarly, despite their large different in computation time. However, we do see that ECOCs outperformed OCs in the SeqSz test. This is because "length" classification attempts to distinguish between a number of very similar categories.

Generally, we can expect ECOCs to perform well when classes are quite similar because every SVM decision acts as a slightly different “vote,” and more votes allow for greater precision. Conversely, one-vs-all output codes perform well when classes are significantly different, and have no structured relationship.

### 5.2 Decision Tree Characteristics

Decision trees are most effected by the structure and ordering of their decision nodes. Looking at our experiments, Figure 6 gives the breakdown of which decisions nodes produce the most errors.

![Figure 6: Error rates for decision tree nodes](Image)
It follows that there will be a low error rate in the leaf decisions because generally a well-formed decision tree should funnel adjacent classifications to leaf SVM nodes, and have the highest error rate in the root node.

However, the $OpInMix$ and $SeqSz$ experiments cannot utilize the balanced tree structure assumed, and give an evenly distributed error rate across all SVM nodes. In these cases, it is likely that $SeqSz$ would benefit from an unbalanced tree, and $OpInMix$ simply introduces too much noise from instructions which are not members of what is being classified.

Decision trees tend to perform best when the structure most accurately matches the underlying data relationships.

6. CONCLUSION

Of the three classification methods examined, decision trees give the best trade-off between low error rates, performance on low samples, and computation time, but only when the tree structure matches the underlying relationships between classes. One-vs-all and error correcting output codes can achieve similar error rates as decision trees, but at the cost of computation time and a higher number of training samples. Our specific problem of classifying hidden instruction sequences is therefore best tackled by a combination of both approaches, given our application domain.

Important to any SVM classification, though, is the choice of features, a kernel, its parameters, and a regularization penalty. As we see with ECOCs, SVMs allow us to model a problem even when little is known about the underlying data. Crucially, feature selection determines the upper limits of what can be learned and what error rates can be achieved. Features which take advantage of the independence of various inputs to our problem are necessary, and more abstract features are able to generalize much better than more specific category-dependent features. Overfitting can occur easily, however, therefore parameter selection must also be fine-tuned for our specific problem.

7. REFERENCES


