Exploiting Similarity Between Variants to Defeat Malware

“Vilo” Method for Comparing and Searching Binary Programs

Andrew Walenstein
University of Louisiana at Lafayette

Blackhat DC 2007
Outline

Motivation
- Few Families, Many Variants
- The Role of Program Binary Comparisons

Vilo: Program Search Methods
- Feature Comparison Approach
- Weighting and Search

Evaluation
- Evaluation Design
- Performance Evaluation
- Accuracy Evaluation
Variety: The Spice of ALife

According to Microsoft’s data [MSIR2006]:
- 97,924 variants in first half of 2006
  - e.g. 3,320 variants of Win32/Rbot, from 5,706 unique files
- that’s > 22 per hour
Microsoft’s Data [MSIR2006]

Data source:

- Win32/R$bot
- Win32/Banker
- Win32/Hupigon
- Win32/Sdbot
- Win32/Small
- Win32/Bancos
- Win32/Agent
- Remaining Top 25
- Rest

Motivation
Search Methods
Evaluation

a. Few Families, Many Variants
So Few Families, So Many Variants

- Clearly all these are **not** new, built-from-scratch!
  - only a few hundred families typical in 6-month period
    [SISTR2006, MSIR2006]

- Variants thus outnumber families by around 500:1
  - top 7 families account for > 1 out of 2 variants
  - top 25 families account for > 3 out of 4 variants
  - good bet:
    - any new malicious program is a variant of a previous one
Malware Evolution Drivers

- What is driving this explosion of variety?
  - cost of constructing malware
  - reduced cycle time for new signature updates
Malware can be costly to develop from scratch
  - a new family can be a substantial investment in time & effort
  - malware authors wish to protect existing investments

Their **problem**: malware detectors catch their code

Their **solution**: change the code
  - can be minor tweaks to throw off signatures
  - cheaper to modify than to build from scratch
  - changes could also be bug fixes, updates, feature additions
    - i.e. standard software evolution

a. Few Families, Many Variants
Update Rate Driver

- **Malware author problem:** rapid signature updates
  - now: daily, sometimes even hourly

- **Their solution:** update frequently
  - can expect signature update rate to pace evolution
    - i.e.: \( \text{rate(malware\_evolution)} \propto \text{rate(signature\_updates)} \)
    - mutation rate increasing to match signature update rates
Impact of Variation on Malware Defense

- Adds layer of complication
  - defense was bad enough before variant flood
  - now malware is a constantly changing target

- Need: systematic ways of coping with variations
  - otherwise rapid evolution becomes DOS attack
  - i.e. flood the limited pool of anti-malware researchers
Why Does Variation Even Work?

- We know most variants differ only slightly
  - shouldn’t this be a significant attack weakness?
- Seems ripe for a counter-attack:
  - AV community has plenty of past samples
  - often only minor changes are made between variants
  - shouldn’t smaller changes = easier detection?
- What is needed:
  - methods for comparing programs to previous ones
    - i.e. ways of searching for matching programs
    - i.e., program similarity measures
Uses for Program Similarity

Measures

- Suppose we had a suitable measure
  - it can compare whole program binaries
  - it is insensitive to minor tweaks and changes

- What might be done with it?

- Two possibilities:
  - automated defenses (?)
    - minor tweaks currently slip past automated defenses
  - support tools for anti-malware researchers
    - high numbers of variants creates burdens on analysts
    - they spend greater fraction of time on already-known threats

b. The Role of Binary Program Comparisons
Current Analyst Scenario

Analyst needs to:

- **Establish malware family**
  - minimal organization-wide resources to consult
  - heavy reliance on past experience, Google

- **Find differences affecting signature matching**
  - ad hoc discovery utilizing manual inspection

- **Figure out how to update the signatures**
  - manual discovery of differences

- **Look for familial similarities**
  - do not want new signature for every variant
  - without whole-family comparison, can miss commonalities
Future Analyst Scenario

Scenario from the future:

- **New unknown sample arrives**
- **Closely related samples are retrieved automatically**
  - analyst need not have seen the family before
- **Associated signatures & documentation are recalled**
  - past efforts are quickly leveraged (organizational knowledge)
- **Analysis of differences highlights changed parts**
  - allows analyst to quickly focus on how to fix signatures
- **Analysis of similarities highlights common features**
  - helps analyst determine how to create generic signatures
Impact to Analyst Scenario

- Direct impact on anti-malware business
  - comparisons help for vast majority of new samples
    - is a critical part of infrastructure, workflow
  - benefits:
    - reduces time to signature release
    - improves detection rates
    - gives team more time to attend to high priority issues
Future Automated Detection Scenario?

Scenario from the future:

- New sample arrives
- It is compared against a database of known malware
- Too similar to existing malware sample?
  - it is filtered
  - what valid program is 99% Win32.Bagle?
- System preemptively defends against close family members
OK, But How?

- The question is: how to compare programs binaries?
- Three key comparison issues considered:
  - Sensitivity of comparison to minor changes
    - adding single C instruction can changed all jump targets
    - reordering statements or procedures
  - Dealing with common code
    - e.g. common libraries, compiler-inserted code
  - Simplicity of analysis method
    - efficiency is always an issue
    - wish to avoid costly analysis like control flow graph extraction

Vilo approach to program comparison
Outline

**Motivation**
- Few Families, Many Variants
- The Role of Program Binary Comparisons

**Vilo: Program Search Methods**
- Feature Comparison Approach
- Weighting and Search

**Evaluation**
- Evaluation Design
- Performance Evaluation
- Accuracy Evaluation
A Program Comparison Approach

- Adaptation of text search and analysis techniques

- Three key ideas underlying the approach:
  - Base similarity comparison on matching code “features”
    - use whole-program comparison, i.e. comprehensive sets
  - Vector model for comparison
    - fast, easy to calculate
  - Statistical weighting for features
    - automatic filtering of “uninteresting” features

- Additional focus: code similarity
  - particular focus is when minor changes are made
  - then it’s important to select the right features

Motivation  Search Methods  Evaluation

a. Feature Comparison Approach
Feature Comparison Approach

- Comparison is based on some set of features

**FEATURES**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Chair 1</th>
<th>Chair 2</th>
<th>Chair 3</th>
<th>Chair 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of legs</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>has a back?</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>amount of cushioning</td>
<td>low</td>
<td>none</td>
<td>high</td>
<td>medium</td>
</tr>
<tr>
<td>is black?</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>
Feature Comparison Approach

- Comparison of objects means comparison of whole list of features

- Example
  - Differences: one leg, cushioning
  - Commonalities: has as back, color
Feature Approach Tradeoffs

- Advantages
  - flexibility: use whatever features make sense
  - order insensitivity: ordering is irrelevant
    - unless features are order sensitive
- However: must get the features right
- Question: what features to use for programs?
n-Grams As Features

- n-gram is a sequence of n “characters” in a row
  - n is typically 2 or 3
  - “characters” can be defined as words, letters, etc.
  - characters can be filtered

- Example: 2-grams, lower-cased ASCII text, whitespace filtered
  - for “The cat is in.”
    - th he ec ca at ti is si in
  - for “Is the cat in?”
    - is st th he ec ca at ti in
  - difference between two: si / st
    - commonalities: at, ca, ec, he, in, is, th, ti

Motivation  Search Methods  Evaluation

a. Feature Comparison Approach
n-grams As Features: Tradeoffs

- Advantages
  - relatively insensitive to order permutation
  - simple to extract automatically
  - easy to compare for commonalities, differences

- Disadvantages
  - number of features can be high
  - some sensitivity to ordering
    - sensitivity related to size of $n$
    - if $n$ is high, any change can affect many features
$n$-grams Applied to Programs

- Many ways of defining and selecting “characters”
  - could use raw bytes
  - could use extracted strings
  - could use disassembly text
  - could be a combination of any of the above

- We have used all of these
  - they all do certain things well

- Our focus here: applications to code, specifically
  - not as well studied
  - difficult for malware author to change

- Approach: use abstracted, disassembled program
n-Grams Using Abstracted Assembly

Many ways to encode assembly
- raw assembly could work
  - convert directly as in text retrieval
- main problem: sensitivity to change
  - inserted instruction changes branch targets
  - data changes, register swaps, all can be unimportant

Approach: use only the operations as characters
- “noise” in the operands do not affect the match
- cannot match on data
- but captures something of the program essence
**n-Grams Encoding of Operations**

### 2-gram tally

<table>
<thead>
<tr>
<th>2-gram</th>
<th>tally</th>
</tr>
</thead>
<tbody>
<tr>
<td>push_mov</td>
<td>1 1</td>
</tr>
<tr>
<td>mov_mov</td>
<td>1</td>
</tr>
<tr>
<td>mov_push</td>
<td>1</td>
</tr>
<tr>
<td>push_cltd</td>
<td>1</td>
</tr>
<tr>
<td>cltd_push</td>
<td>1</td>
</tr>
</tbody>
</table>

**55**
- push ebp
- mov $0x11,eax

**b8 11 00 00 00**
- mov esp,ebp
- push edi

**89 e5**
- cltd

**57**
- push esi

**99**
- mov $0x11,0xffe4(ebp)

**56**
- cltd

**c7 45 e4 11 00 00 00**
- mov $0x11,0xffe4(ebp)
Reducing Order Sensitivity: $n$-Perms

- $n$-grams are sequence specific
  - $n$-grams over operation sequences are sensitive to ordering
  - modifications may change the orderings
    - e.g. permuting order of non-dependent statements

- Defined $n$-perms as variants of $n$-grams
  - difference: match does not consider order of characters
    - “the” matches “teh” matches “eth”
### $n$-Perm Encoding of Operations

| 55 | b8 11 00 00 00 | push ebp |
| 89 | e5           | mov $0x11,eax |
| 57 |             | mov esp,ebp |
| 99 |             | push edi   |
| 56 |             | cltd       |
| c7 45 e4 11 00 00 00 | push esi |
|     | mov         | $0x11,0xffe4(ebp) |

#### 2-perm Tally

<table>
<thead>
<tr>
<th>2-perm</th>
<th>tally</th>
</tr>
</thead>
<tbody>
<tr>
<td>push_mov</td>
<td>1 1 1</td>
</tr>
<tr>
<td>mov_mov</td>
<td>1</td>
</tr>
<tr>
<td>push_cltd</td>
<td>1 1</td>
</tr>
</tbody>
</table>

---

**Motivation**

- Search Methods
- Evaluation

---

| 04/01/2007 | Blackhat DC |  
Walenstein | Exploiting Similarity | Between Variants |  
Motivation | Search Methods | Evaluation |  
a. Feature Comparison Approach
Differences Between Grams/Perms

- Advantages of \( n \)-perms over \( n \)-grams
  - number of features is reduced (for equivalent \( n \))
    - “the” and “teh” are distinct features under \( n \)-grams
  - reduce sensitivity to order changes
    - e.g., code permutations, such as statement reordering

- Disadvantages
  - false matches more likely for any given \( n \)
    - must use larger \( n \) to reduce false matches

- \( n \)-perms appear to work well on code [PHYLO2005]
  - part of a pending patent
Vector-Based Similarity Calculation

- Each feature is treated as a dimension
  - programs are summarized as a vector of feature counts
    - i.e. mapped to points in a multi-dimensional space
  - e.g. $\text{num\_legs} = [5, 1, 2, 1]$
Vector Representation of Assembly

Frequency counts turned into vector

- \[ [3 \ 1 \ 2] \]
Vectors Comparison

- Vectors compared by measuring their cosine angle
  - think: high similarity = arrows pointing in the same direction
  - e.g., $v_1 = [3, 1, 2]$ compared to $v_2 = [4, 0, 5]$

$$\frac{v_1 \cdot v_2}{|v_1| |v_2|} = \frac{3 \times 4 + 1 \times 0 + 2 \times 5}{\sqrt{3^2 + 1^2 + 2^2} \sqrt{4^2 + 0^2 + 5^2}} = 0.918$$
Feature Interestingness

- Not all features are equally interesting
  - e.g., standard function epilogues
    - occur many times, are in essentially all programs
  - e.g., standard linked-in features
    - startup and exit code, standard libraries
  - such features should not be as important for similarity
    - may be interesting to know two viruses use same libraries
    - but do not want similarity scores to reflect primarily that

- Needed:
  - a way to adjust how important the features are
  - and do not wish to manually or statically do this
Solution: Statistical Weighting

- Idea comes from text retrieval’s “TF x IDF” scheme
  - idea: weight features according to inverse of commonality
  - common features = not interesting

- Approach:
  - select a corpus or database of malware
  - for each feature, count the number of samples it appears in
  - weight feature counts by dividing by the feature frequencies
    - e.g., if A appears in 10 out of 100, weight A counts by 1/10
    - (a variety of formulas can be used too)
Weighting Example

- Given two vectors for worms from a database of 10
  - \( \text{worm}_1: [3 \ 4 \ 2 \ 1] \)
  - \( \text{worm}_2: [4 \ 5 \ 1 \ 0] \)
  - cosine similarity: \( \text{sim}(\text{worm}_1,\text{worm}_2) = .958 \)

- Weighting the feature count vectors
  - feature counts: \( [9 \ 8 \ 3 \ 2] \)
    - i.e., feature 1 is in 9 out of 10 samples
  - \( \text{weighted}_1: [3/9 \ 4/8 \ 2/3 \ 1/2] = [0.33 \ 0.25 \ 0.66 \ 0.50] \)
  - \( \text{weighted}_2: [4/9 \ 5/8 \ 1/3 \ 0/2] = [0.44 \ 0.63 \ 0.33 \ 0.00] \)
  - cosine similarity: \( \text{sim}(\text{weighted}_1, \text{weighted}_2) = .795 \)

- First two features are very common
  - weighted versions decrease their relative importance
Advantages of Weighting Scheme

- The scheme automatically scales common code
  - e.g., when same compiler used by multiple worms

- Weights can be automatically adjusted
  - can be incrementally calculated when adding new samples

- Can pre-weight the database
  - import standard library code as samples
  - initialize their feature counts with high values
    - serves to de-emphasize known irrelevant features
    - can be used to remove problem false matches
With similarity function, one can search a database

- collect together some known malware
- load the database with feature count vectors from these
- extract feature count vector from unknown program U
- for every vector in database
  - calculate weighted cosine similarity to U
- sort list of similarities

Result: ranked list of matches
Summary of Approach

- **Simplicity**
  - automatic way of extracting features
  - easy arithmetic for vector scaling and comparison
  - needs disassembly, but nothing else
  - compare: using control-flow-graphs or semantic graphs

- **Insensitivity to program modifications**
  - by design, is Insensitive to sequence
    - e.g. code motion and permutations
      - permutation affects only handful of features
      - particularly when using n-perms
  - compare: sequence-based approaches
    - e.g. longest common subsequence sensitive to block moves
Summary of Approach

- Ability to filter “uninteresting” features
  - automatic, based on corpus of samples
  - allows specific filtering without manually tuning features

- Flexibility
  - mix-and-match feature types
    - $n$-grams/perms, strings, bytes, etc.
Outline

- Motivation
  - Few Families, Many Variants
  - The Role of Program Binary Comparisons
- Vilo: Program Search Methods
  - Feature Comparison Approach
  - Weighting and Search

Evaluation

- Evaluation Design
- Performance Evaluation
- Accuracy Evaluation
How Well Does the Approach Work?

- Dimensions to evaluate
  - Does the search scale?
    - Can we search against useful sized databases?
  - Is accuracy good?
    - Will it catch minor variants?
    - How frequently will false positives occur?
- Two studies conducted to shed light on these
Apparatus

- Implementation of Vilo approach
  - core search implemented in C
    - reads database of feature count vectors
    - queries are other feature count vectors
    - returns ranked list of matches

- Implemented as an independent component
  - component part of “search-as-a-service” environment
  - runs as daemon under Linux
  - prototype web-based portal under development
Implementation Specifics

- For building a database:
  - disassembly currently using **objdump** (GNU binutils)
    - but have used IDA Pro™, but with some limitations
    - n.b., the programs must not be encrypted or packed
  - 10-perms used for our tests

- For querying:
  - feature count vector extracted same way
  - vector is sent to server, and results are read

- Interfaces:
  - server components and command line tools
  - JSP-based wrapper / interface
## Samples matching the uploaded file

<table>
<thead>
<tr>
<th>Score</th>
<th>Size</th>
<th>Matched Sample Info</th>
<th>md5</th>
<th>compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>91,204</td>
<td>sample/Klez-H</td>
<td>74e3e172fe55e10b36078c481b514a2d</td>
<td>PE strings asm</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ClamAV:</strong> Worm.Klez.H</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>BitDefender:</strong> Win32.Klez.H@mm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>95,800</td>
<td>Worm.Klez.H-I-Worm.Klez.i</td>
<td>543c358d51a949d6584f568bc3ac465b</td>
<td>PE strings asm</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ClamAV:</strong> Worm.Klez.H</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>BitDefender:</strong> Win32.Klez.l@mm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>90,099</td>
<td>20050307-Worm-Klez-H-20050207-162358-bat</td>
<td>105958b332da020bb7f60eaa5f2faf25</td>
<td>PE strings asm</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ClamAV:</strong> Worm.Klez.H</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>BitDefender:</strong> Win32.Klez.H@mm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

04/01/2007 | Blackhat DC | Walenstein | Exploiting Similarity Between Variants

Motivation | Search Methods | Evaluation

a. Evaluation Design
### Comparing PE Information

<table>
<thead>
<tr>
<th>Uploaded File</th>
<th>Matched File</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Identity</strong></td>
<td><strong>Identity</strong></td>
</tr>
<tr>
<td>Ident</td>
<td>Worm.Klez.E-1</td>
</tr>
<tr>
<td>Size</td>
<td>61,440</td>
</tr>
<tr>
<td><strong>Sections</strong></td>
<td><strong>Sections</strong></td>
</tr>
<tr>
<td>Name</td>
<td>Start</td>
</tr>
<tr>
<td>.text</td>
<td>1000</td>
</tr>
<tr>
<td>.rcdata</td>
<td>d000</td>
</tr>
<tr>
<td>.data</td>
<td>f000</td>
</tr>
<tr>
<td>.rsrcc</td>
<td>14000</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Imports</strong></td>
<td><strong>Imports</strong></td>
</tr>
<tr>
<td>KERNEL32.dll</td>
<td>KERNEL32.dll</td>
</tr>
<tr>
<td>ADVAPI32.dll</td>
<td>ADVAPI32.dll</td>
</tr>
<tr>
<td>WS2_32.dll</td>
<td>WS2_32.dll</td>
</tr>
<tr>
<td>MPR.dll</td>
<td>MPR.dll</td>
</tr>
<tr>
<td><strong>Evaluation Design</strong></td>
<td></td>
</tr>
</tbody>
</table>

Motivation  
Search Methods  
Evaluation  
a. Evaluation Design
## Comparing Strings

### String Comparison

<table>
<thead>
<tr>
<th>Strings only in uploaded: &quot;worm-Klez-H-090390.001&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not including dups: 3</td>
</tr>
<tr>
<td>Dups included: 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strings only in matched: &quot;sample/Klez-H&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not including dups: 204</td>
</tr>
<tr>
<td>Dups included: 507</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strings Common to Both</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not including dups: 271</td>
</tr>
<tr>
<td>Dups included: 1116</td>
</tr>
</tbody>
</table>

### Strings in Uploaded file "worm-Klez-H-090390.001" only

<table>
<thead>
<tr>
<th>Number</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$ !*</td>
</tr>
<tr>
<td>2</td>
<td>01606</td>
</tr>
<tr>
<td>6</td>
<td>81606</td>
</tr>
</tbody>
</table>

### Strings in Matched file "sample/Klez-H" only

<table>
<thead>
<tr>
<th>Number</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>!JS?/</td>
</tr>
<tr>
<td>2</td>
<td>#Eki]QS</td>
</tr>
<tr>
<td>2</td>
<td>#MWEs]SE</td>
</tr>
<tr>
<td>2</td>
<td>%SGE]cEhMKwE</td>
</tr>
<tr>
<td>2</td>
<td>%oGMgEi</td>
</tr>
<tr>
<td>2</td>
<td>'kEsg</td>
</tr>
<tr>
<td>2</td>
<td>'kEGU]IkQ</td>
</tr>
<tr>
<td>4</td>
<td>'kMScEek</td>
</tr>
<tr>
<td>2</td>
<td>)E 'IK/k]c]WEAE</td>
</tr>
<tr>
<td>4</td>
<td>)ElekE</td>
</tr>
<tr>
<td>2</td>
<td>)Egg]Ski?</td>
</tr>
<tr>
<td>2</td>
<td>)EkoEk</td>
</tr>
<tr>
<td>2</td>
<td>)H]Sg</td>
</tr>
<tr>
<td>6</td>
<td>)QGgaMkE7</td>
</tr>
</tbody>
</table>

(Click on HELP for an explanation of this page)
Comparing Disassembly

Motivation
Search Methods
Evaluation

a. Evaluation Design
Basic Performance Evaluation

- Query time is a critical performance issue
  - must be able to query against large enough database
  - should be interactive even when many samples involved

- Evaluation method:
  - load database with sample sets of different sizes
  - average times for 200 randomly selected samples
  - measure time and memory usage
    - query time only
    - not transmission and parsing overheads
Subject / Data Set

- Data was generated
  - did not have access to thousands of authentic variants

- Group properties of the dataset are important
  - query speed affected by sample sizes
  - memory use is affected by
    - number of families
    - evolution rate between variants
Data Set Construction / Properties

- Projected from collection of authentic samples
  - 542 samples collected from mail server and web
  - primarily worms and Trojans (Win32)

- Projection method
  - size of created samples projected from authentic distribution
  - 1 out of 2 are modified versions of another
  - evolution rate between versions is half a % difference
    - in practice, authentic variants are often much less different
Results: Memory & CPU Usage

Database Size

- Avg Query Time (milliseconds)
- Memory Usage (MB)
Accuracy Test Design

- Two error classes:
  - false negative: a good match was not reported
  - false positive: a match reported is not a good match
  - “good” match: known to be related or close in some way

- Evaluation method:
  - load database with samples
    - simulating typical menagerie of malice
    - derivation relationships known between samples
  - two query sessions using similarity threshold of .100 and .002
    - nothing returned less than these thresholds

- Measures:
  - precision and recall

Motivation  Search Methods  Evaluation

C. Accuracy Evaluation
Data Set Construction

- Data set is generated
  - 264 samples of Win32 malware selected from first
    - all are from top-25 families in 2006, as named by Microsoft [MSIR2006]
    - 36 of these identified as family constructed using construction kit
  - 202 variants constructed using construction kit in forensic environment
    - known to be derivatives by construction
    - related to the 36 collected from the wild
  - 466 samples total

[04/01/2007 | Blackhat DC | Walenstein Exploiting Similarity Between Variants]
Results and Discussion

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Mean Precision</th>
<th>Mean Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>.002</td>
<td>0.79</td>
<td>1.00</td>
</tr>
<tr>
<td>.100</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Limited test due to limitations of database
- Optimum threshold for data set is at .100
  - no point increasing threshold, since:
    - no fewer false positives (precision is 100%)
    - only fewer matches (recall drops)
  - still a small number
Conclusions

- Assembly-based vector matching is promising
  - simple and automatic
  - scalable to databases of 10s of thousands
    - at least efficient for interactive matching, such as in triage
  - designed to account for expected variation
    - via selection of whole-program feature matching
    - due to selection of feature types
  - good preliminary results
  - may be suitable for automated detection
References


  http://www.symantec.com/enterprise/threatreport/index.jsp
Acknowledgements

Current Members of the Software Research Laboratory

- Arun Lakhotia, Director
- Michael Venable, Research Associate
- Ph.D. Students
  - Mohamed R. Chouchane
  - Md.-Enam Karim
- M.Sc. Students
  - Matthew Hayes
  - Chris Thompson

Recent Graduates

- Aditya Kapoor, McAfee
- Eric Uday Kumar, Authentium
- Rachit Mathur, McAfee