Exploiting Similarity Between Variants to Defeat Malware

“Vilo” Method for Comparing and Searching Binary Programs

Andrew Walenstein

University of Louisiana at Lafaytte

Blackhat DC 2007

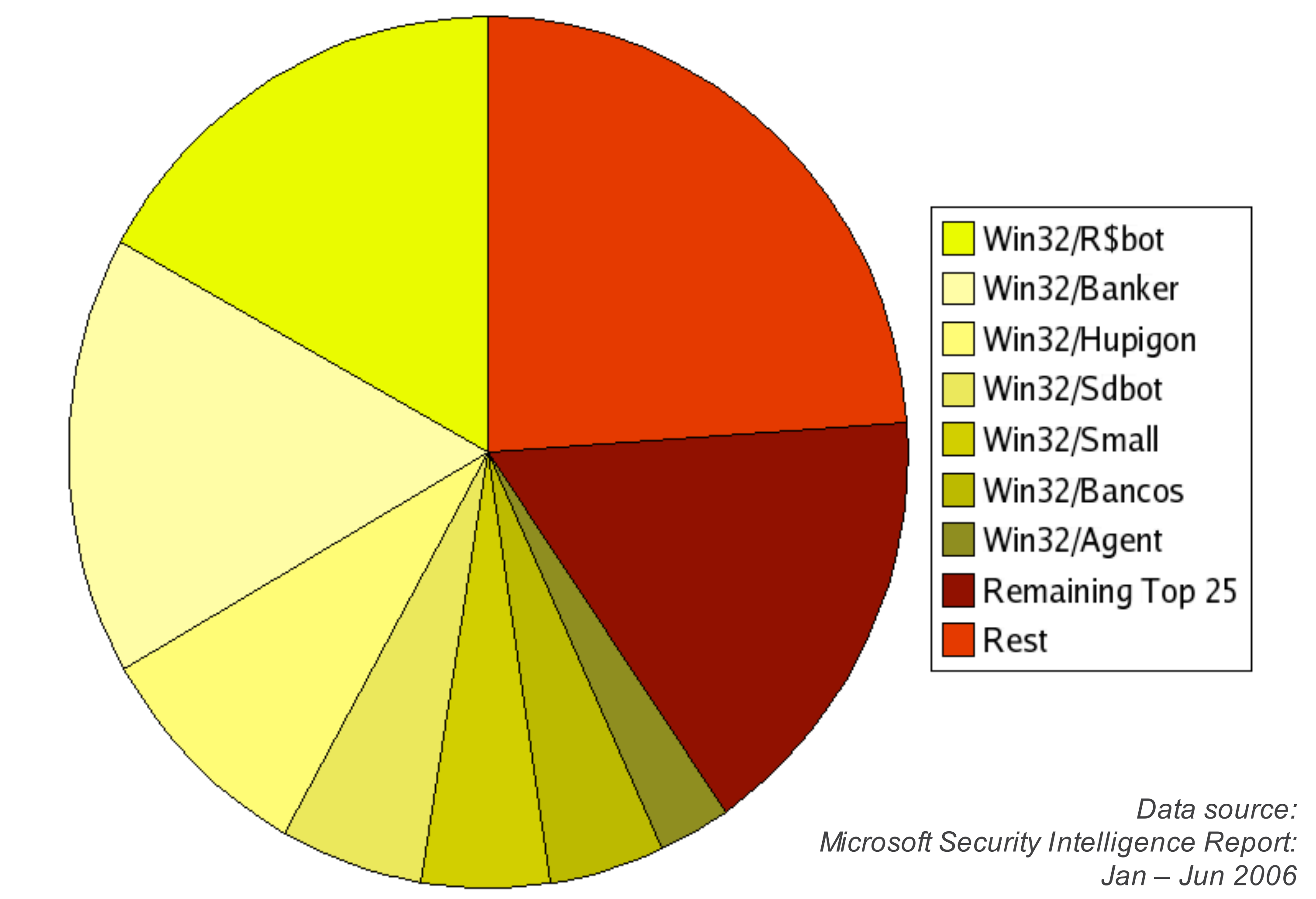
Outline

- 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:
Microsoft Security Intelligence Report:
Jan – Jun 2006

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 Construction Cost Drivers

- 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

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.: *rate(malware_evolution)* ∝ *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

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

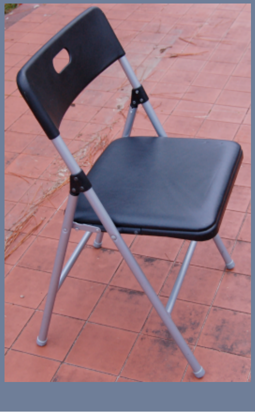
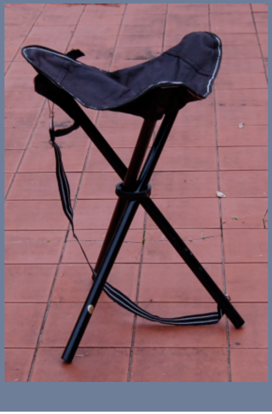
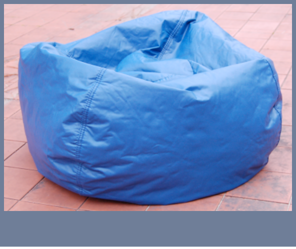
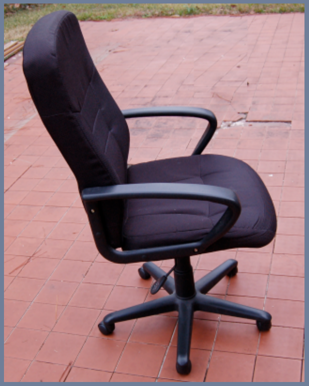
- Motivation
  - Few Families, Many Variants
  - The Role of Program Binary Comparisons
- Vilo: Program Search Methods
  - Feature Comparison Approach
  - Weighting and Search
- Evaluation
  - Evaluation Design
  - Performance Evaluation
  - Accuracy Evaluation

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 its important to select the right features

Feature Comparison Approach

- Comparison is based on some set of features

FEATURES				
number of legs	4	3	0	5
has a back?	Y	N	N	Y
amount of cushioning	low	none	high	medium
is black?	Y	Y	N	Y

Feature Comparison Approach

- Comparison of objects means comparison of whole list of features

vs

- 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**

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

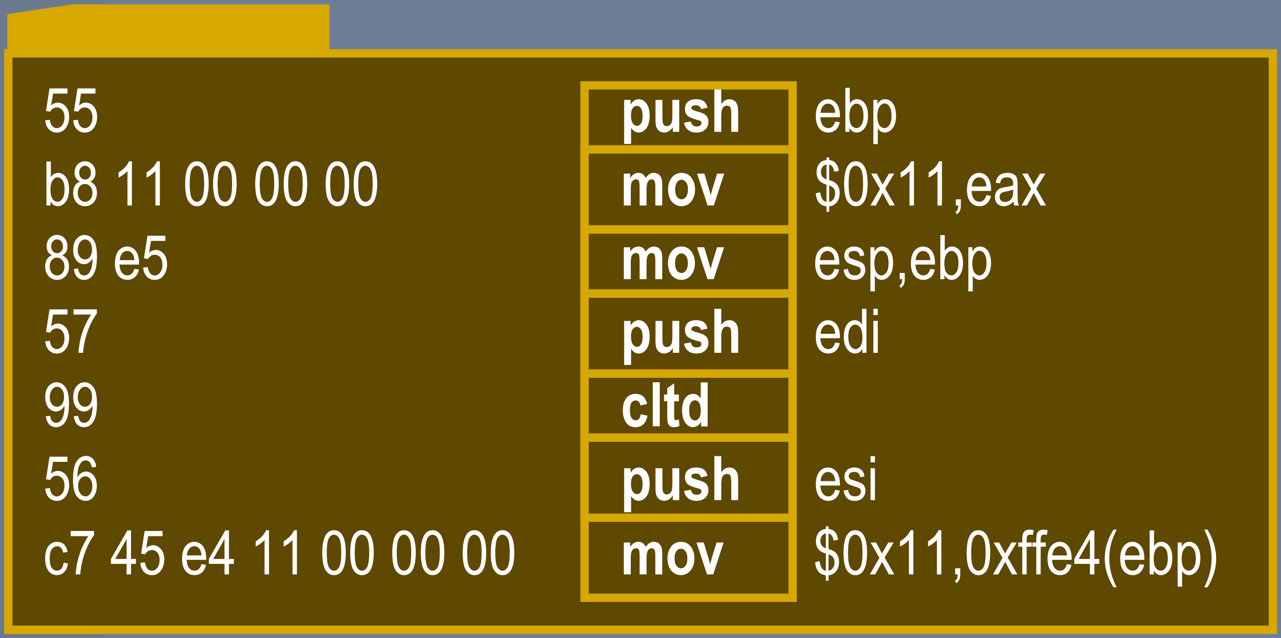
n-grams Applied to Programs

- 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



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

2-gram	tally
push_mov	1 1
mov_mov	1
mov_push	1
push_cltd	1
cltd_push	1

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

2-perm	tally
push_mov	1 1 1
mov_mov	1
push_cltd	1 1

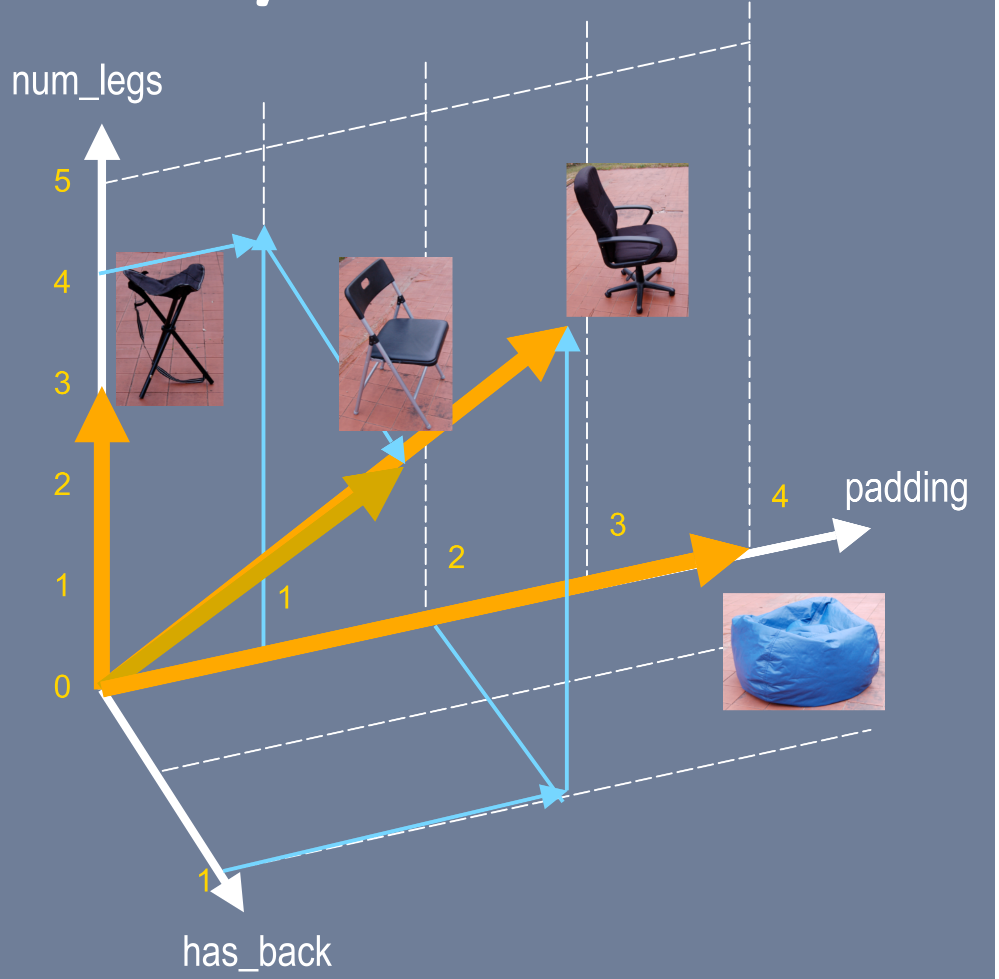
Differences Between Grams/Perms

- 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.

= [5 1 2 1]



Vector Representation of Assembly

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

2-perm	freq
push_mov	3
mov_mov	1
push_cltd	2

- 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]

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

Feature Interestingness

- Not all features are equally interesting
  - e.g., standard function epilogs
    - 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
  - *$worm_1$*: [3 4 2 1]
  - *$worm_{2:}$* [4 5 1 0]
  - cosine similarity: *$sim(worm_1,worm_2)$* = .958
- Weighting the feature count vectors
  - feature counts: [9 8 3 2]
    - i.e., feature 1 is in 9 out of 10 samples
  - *$weighted_1$*: [3/9 4/8 2/3 1/2] = [.33 .25 .66 .50]
  - *$weighted_2$*: [4/9 5/8 1/3 0/2] = [.44 .63 .33 .00]
  - cosine similarity: sim(weighted1, weighted2) = .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

Searching

- 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

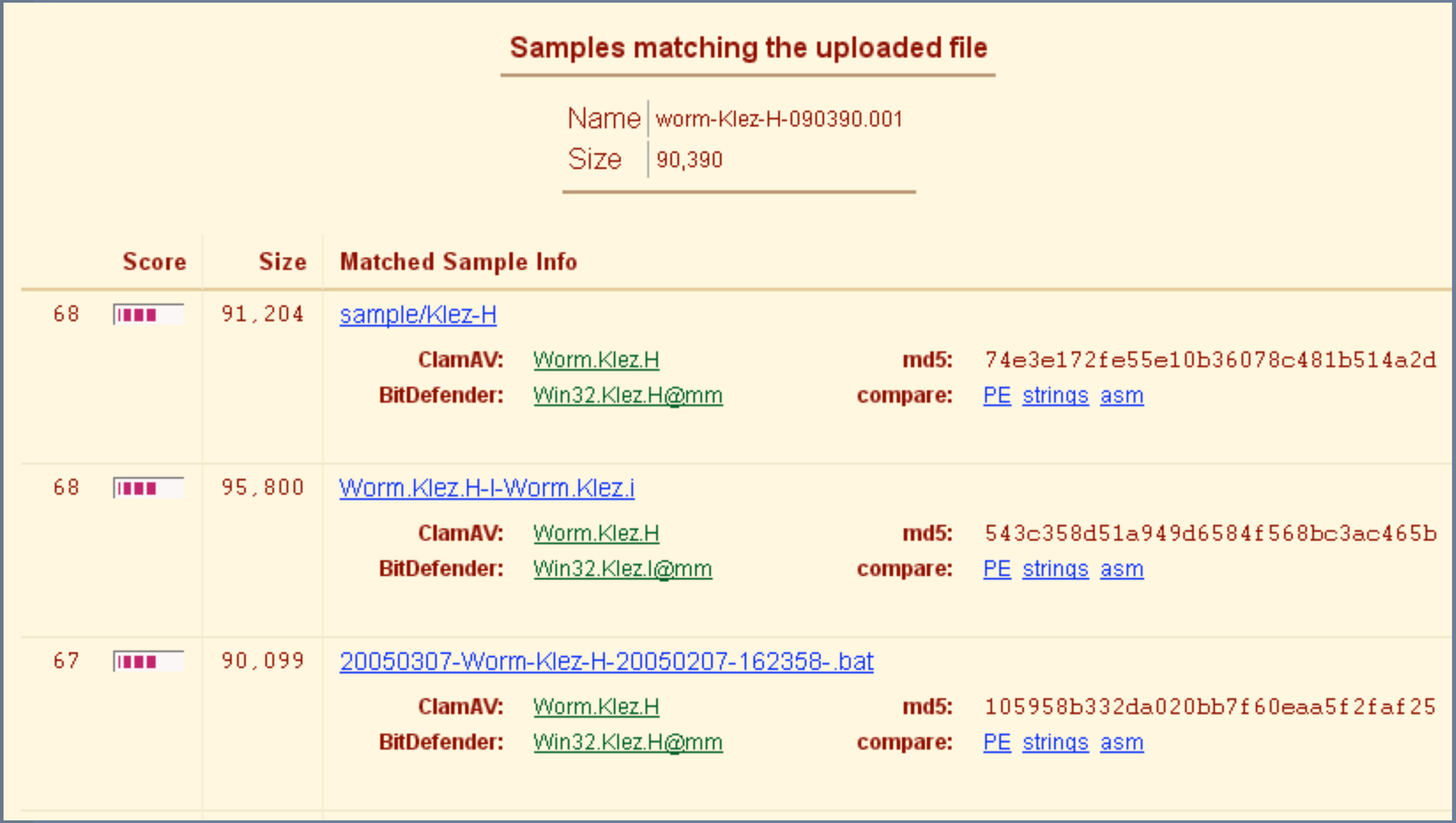
Apparatus

- 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

Matching

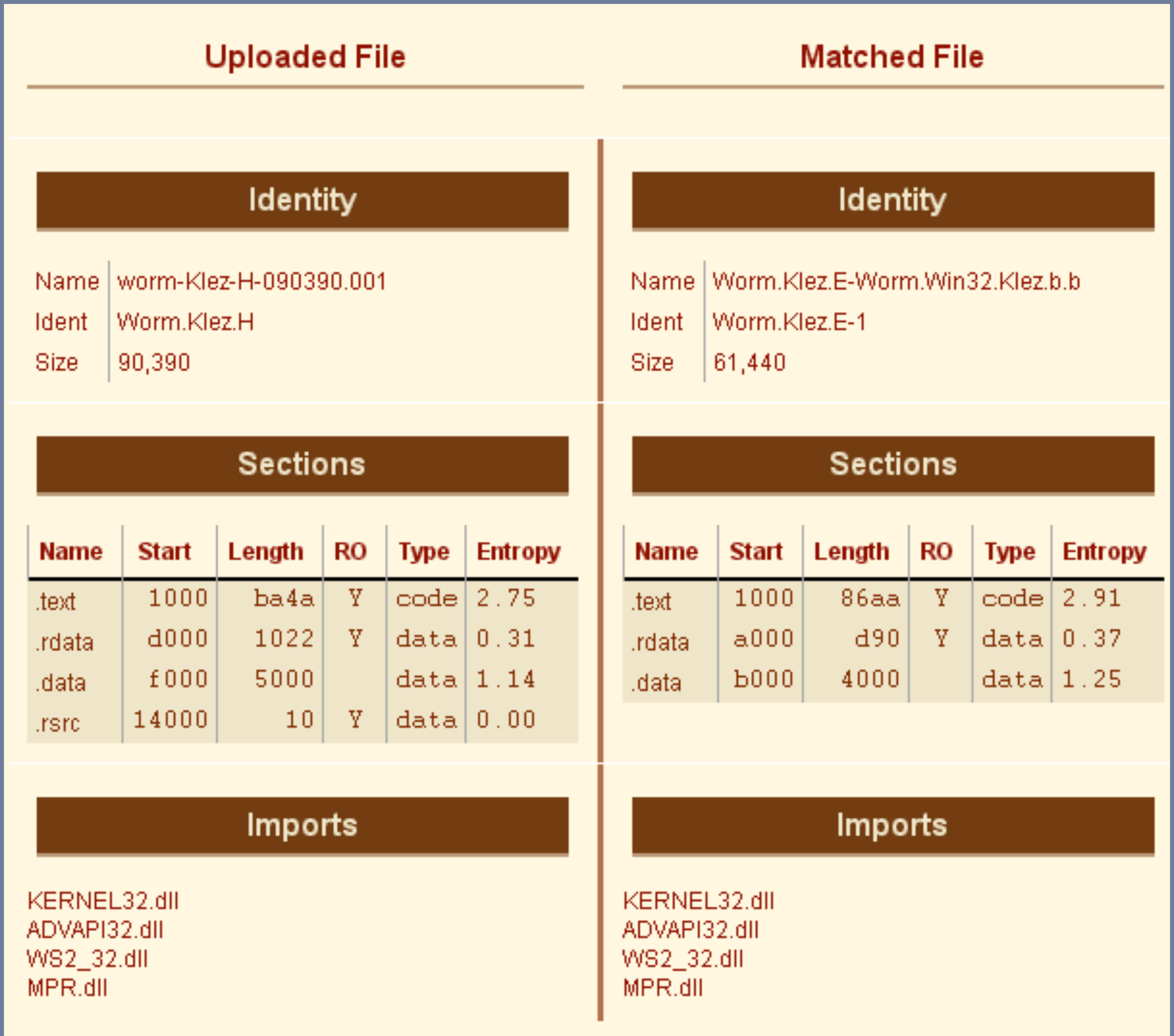


Samples matching the uploaded file

Name	worm-Klez-H-090390.001
Size	90,390

Score		Size	Matched Sample Info
68	▮▮▮▮	91,204	sample/Klez-H **ClamAV:** Worm.Klez.H **md5:** 74e3e172fe55e10b36078c481b514a2d **BitDefender:** Win32.Klez.H@mm **compare:** PE strings asm
68	▮▮▮▮	95,800	Worm.Klez.H-I-Worm.Klez.i **ClamAV:** Worm.Klez.H **md5:** 543c358d51a949d6584f568bc3ac465b **BitDefender:** Win32.Klez.I@mm **compare:** PE strings asm
67	▮▮▮▮	90,099	20050307-Worm-Klez-H-20050207-162358-.bat **ClamAV:** Worm.Klez.H **md5:** 105958b332da020bb7f60eaa5f2faf25 **BitDefender:** Win32.Klez.H@mm **compare:** PE strings asm

Comparing PE Information



Uploaded File

Identity

Name	worm-Klez-H-090390.001
Ident	Worm.Klez.H
Size	90,390

Sections

Name	Start	Length	RO	Type	Entropy
.text	1000	ba4a	Y	code	2.75
.rdata	d000	1022	Y	data	0.31
.data	f000	5000		data	1.14
.rsrc	14000	10	Y	data	0.00

Imports

KERNEL32.dll
ADVAPI32.dll
WS2_32.dll
MPR.dll

Matched File

Identity

Name	Worm.Klez.E-Worm.Win32.Klez.b.b
Ident	Worm.Klez.E-1
Size	61,440

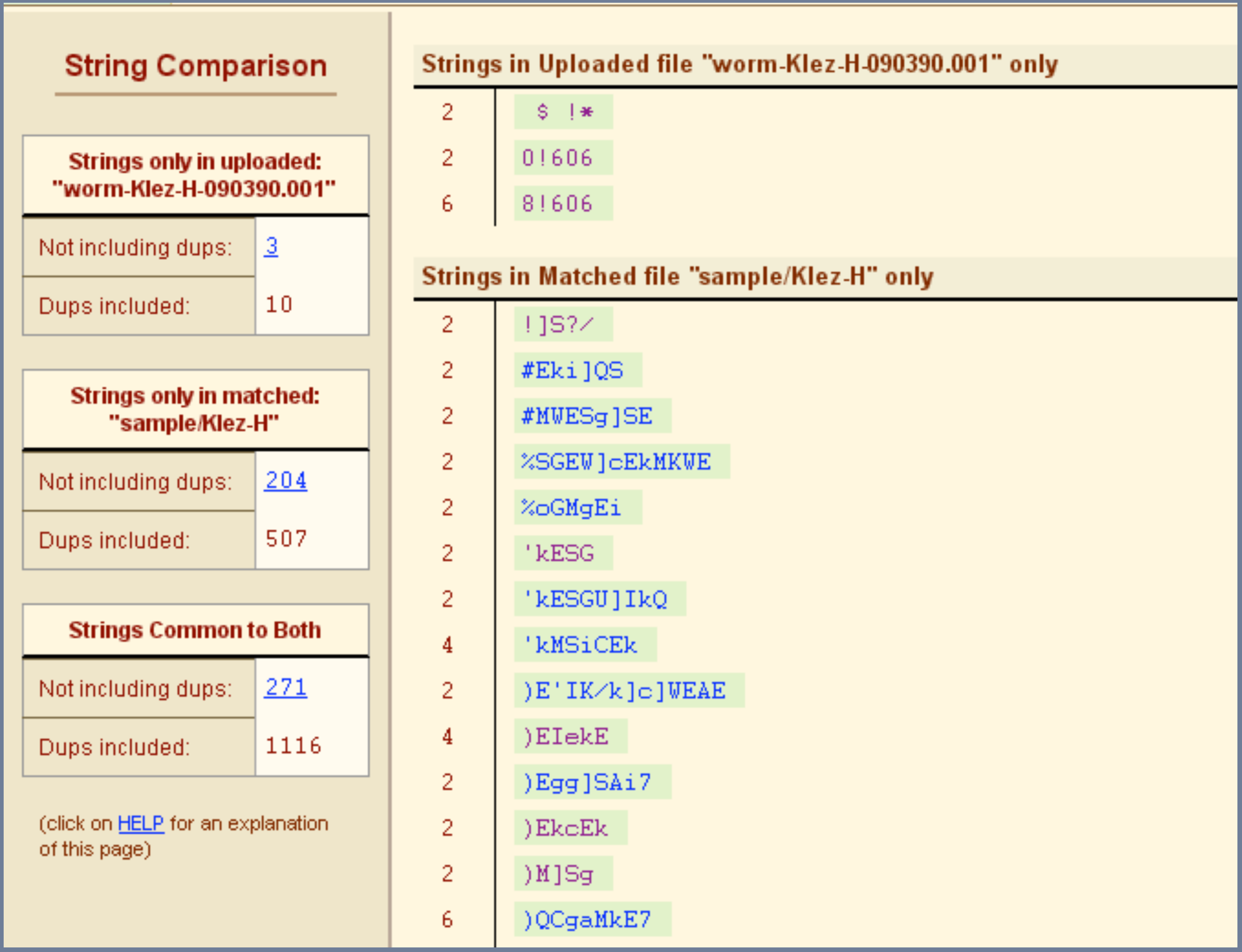
Sections

Name	Start	Length	RO	Type	Entropy
.text	1000	86aa	Y	code	2.91
.rdata	a000	d90	Y	data	0.37
.data	b000	4000		data	1.25

Imports

KERNEL32.dll
ADVAPI32.dll
WS2_32.dll
MPR.dll

Comparing Strings



String Comparison

Strings only in uploaded: "worm-Klez-H-090390.001"	
Not including dups:	3
Dups included:	10

Strings only in matched: "sample/Klez-H"	
Not including dups:	204
Dups included:	507

Strings Common to Both	
Not including dups:	271
Dups included:	1116

(click on HELP for an explanation of this page)

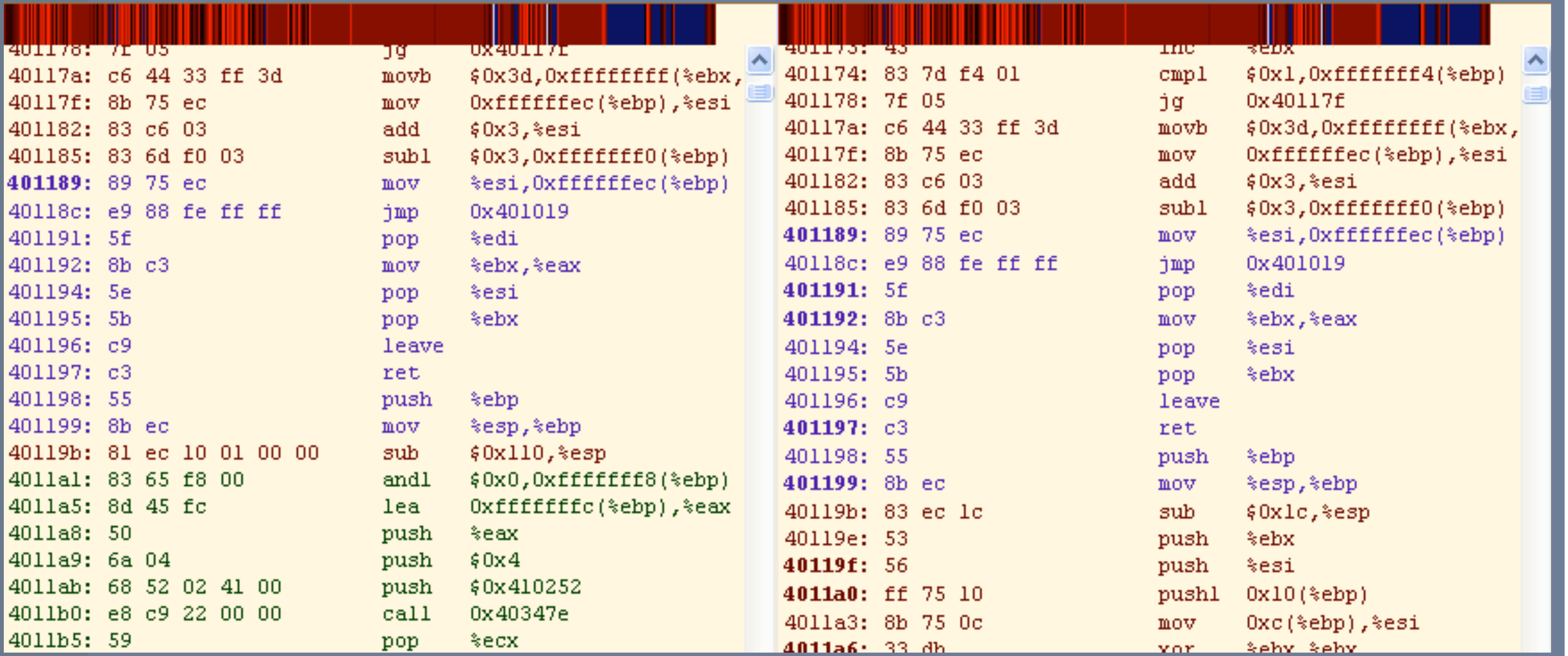
Strings in Uploaded file "worm-Klez-H-090390.001" only

2	$!*
2	0!606
6	8!606

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

2	!]S?/
2	#Eki]QS
2	#MWESg]SE
2	%SGEW]cEkMKWE
2	%oGMgEi
2	'kESG
2	'kESGU]IkQ
4	'kMSiCEk
2	)E'IK/k]c]WEAE
4	)EIekE
2	)Egg]SAi7
2	)EkcEk
2	)M]Sg
6	)QCgaMkE7

Comparing Disassembly



```
401178: 7f 05                jg     0x40117f
40117a: c6 44 33 ff 3d       movb   $0x3d,0xffffffff(%ebx,
40117f: 8b 75 ec             mov    0xffffffec(%ebp),%esi
401182: 83 c6 03             add    $0x3,%esi
401185: 83 6d f0 03          subl   $0x3,0xfffffff0(%ebp)
401189: 89 75 ec             mov    %esi,0xffffffec(%ebp)
40118c: e9 88 fe ff ff       jmp    0x401019
401191: 5f                   pop    %edi
401192: 8b c3                mov    %ebx,%eax
401194: 5e                   pop    %esi
401195: 5b                   pop    %ebx
401196: c9                   leave
401197: c3                   ret
401198: 55                   push   %ebp
401199: 8b ec                mov    %esp,%ebp
40119b: 81 ec 10 01 00 00    sub    $0x110,%esp
4011a1: 83 65 f8 00          andl   $0x0,0xfffffff8(%ebp)
4011a5: 8d 45 fc             lea    0xfffffffc(%ebp),%eax
4011a8: 50                   push   %eax
4011a9: 6a 04                push   $0x4
4011ab: 68 52 02 41 00       push   $0x410252
4011b0: e8 c9 22 00 00       call   0x40347e
4011b5: 59                   pop    %ecx
```

```
401173: 43                   inc    %ebx
401174: 83 7d f4 01          cmpl   $0x1,0xfffffff4(%ebp)
401178: 7f 05                jg     0x40117f
40117a: c6 44 33 ff 3d       movb   $0x3d,0xffffffff(%ebx,
40117f: 8b 75 ec             mov    0xffffffec(%ebp),%esi
401182: 83 c6 03             add    $0x3,%esi
401185: 83 6d f0 03          subl   $0x3,0xfffffff0(%ebp)
401189: 89 75 ec             mov    %esi,0xffffffec(%ebp)
40118c: e9 88 fe ff ff       jmp    0x401019
401191: 5f                   pop    %edi
401192: 8b c3                mov    %ebx,%eax
401194: 5e                   pop    %esi
401195: 5b                   pop    %ebx
401196: c9                   leave
401197: c3                   ret
401198: 55                   push   %ebp
401199: 8b ec                mov    %esp,%ebp
40119b: 83 ec 1c             sub    $0x1c,%esp
40119e: 53                   push   %ebx
40119f: 56                   push   %esi
4011a0: ff 75 10             pushl  0x10(%ebp)
4011a3: 8b 75 0c             mov    0xc(%ebp),%esi
4011a6: 33 db                xor    %ebx,%ebx
```

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 fo 200 randomly selected samples
  - measure time and memory usage
    - query time only
    - not transmission and parsing overheads

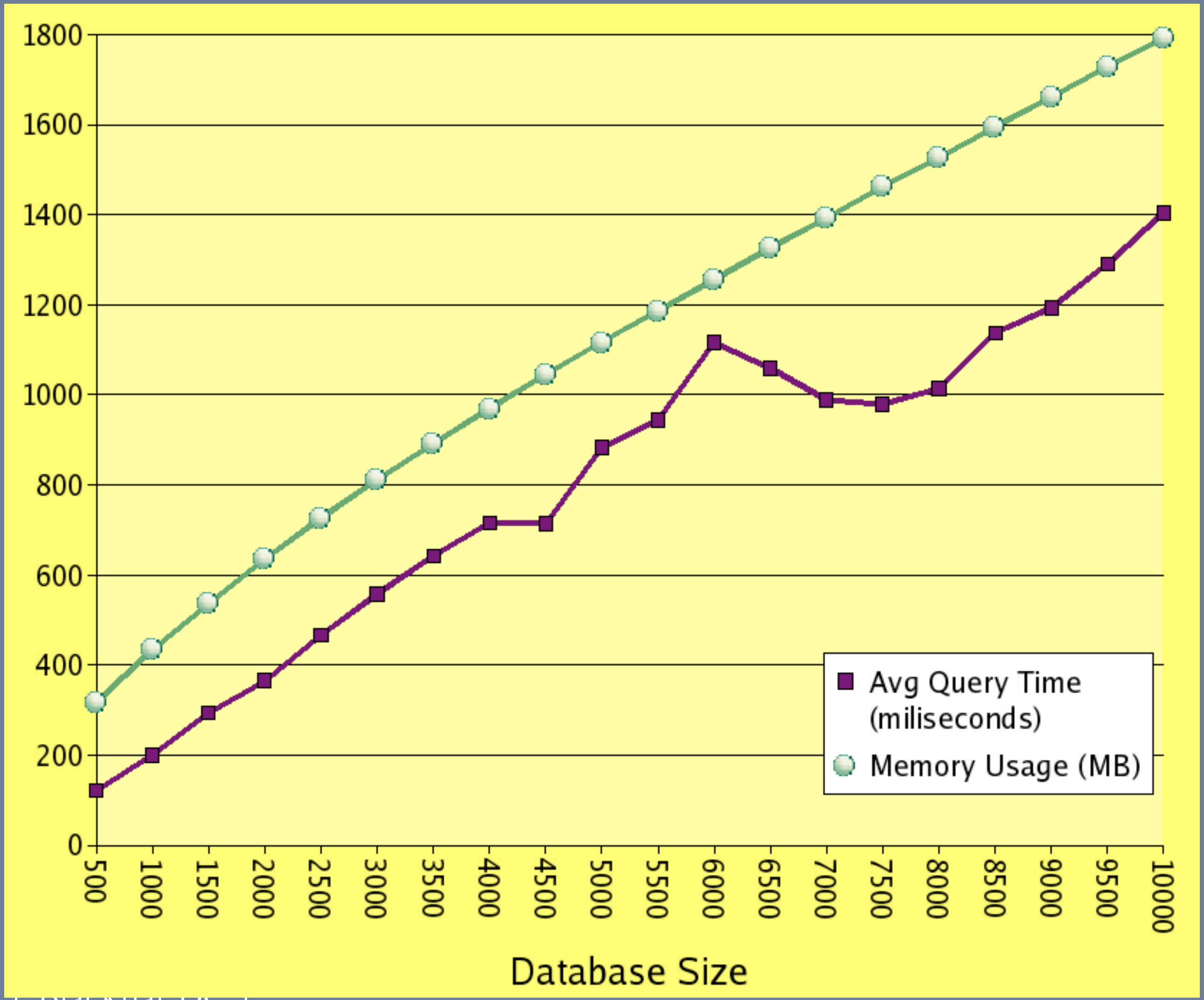
Subject / Data Set

- 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



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

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

Results and Discussion

Threshold	Mean Precision	Mean Recall
.002	0.79	1.00
.100	1.00	1.00

- 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

MSIR2006 Microsoft. ***Microsoft Security Intelligence Report: Jan – Jun 2006***. http://www.microsoft.com/downloads/details.aspx?FamilyId=1C443104-5B3F-4C3A-868E-36A553FE2A02

PHYLO2005 Karim, Md.-E., Walenstein, A., Lakhotia, A., and Parida, L., ***Malware Phylogeny Generation Using Permutations of Code***, Journal in Computer Virology, 1(1), 2005, pp. 13-23.

http://www.springerlink.com/content/u57334818560381

SISTR2006 Symantec, ***Internet Security Threat Report Volume X: September 2006***. http://www.symantec.com/enterprise/threatreport/index.jsp

Acknowledgements

Current Members of the Software Reasearch 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