UNIVERSITY OF
CAMBRIDGE

Rendezvous: A search engine for binary code

Wei Ming Khoo, Alan Mycroft, Ross Anderson
University of Cambridge

CREST Open Workshop on Malware
29 May 2013

Demo: http://www.rendezvousalpha.com

Software reverse engg.

Software RE is tedious, requires expertise

- Decompilers
  - Boomerang, REC Studio 4, Anatomizer, Andromeda, exetoc, desquirr
  - Current state-of-the-art: Hex-Rays, USD$1,160 per license per year + expertise
  - 415 man-hours to decompile 1,500 LoC comprising 8% of code base [VanEmmerik'04]
- Stuxnet
  - Assuming deployed in June 2009, took a year to be discovered, a further 5 months for AV and SCADA experts to decipher the payload

But, code reuse is prevalent

And increasingly so due to advances in software mining and SBSE

- Catalysts include market competitiveness, application complexity, quality of reusable components [Schmidt'99, '00, '06]
- Six open source projects: On average 74% of code base was external [Haefliger'08]
- Sometimes illegally: >250 products found GPL non-compliant, most famously Linksys WRT54G

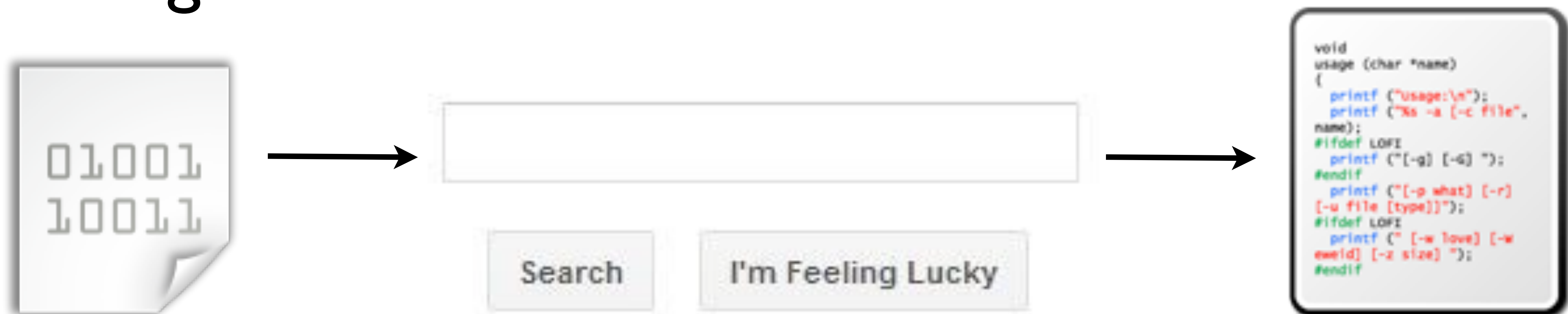
Code reuse in malware

- Malware producers operate very much like corporations
  - Innovative Marketing Ukraine: revenue of about $180 million in 2008, complete with HR dept and call center (Finkle'10)
- ZeuS 2.0.8.9 source code leaked in May 2011 revealed the following FOSS components
  - xterm/key2symucs.c (© April 2001 Markus G. Kuhn, U. Cambridge)
  - UltraVNC circa 2005
  - BEA disassembly engine
  - UCL compression lib 1.0.3
  - Info-zip 2.3.2
  - Mersenne twister PRNG circa 2002 (Matsumoto & Nishimura)
- 94.3% LoC of ZeuS 2.0.8.9 was reused, only 5.7% was new functionality

Proposed solution

Search-based reverse engineering (SBRE)

"Google" it:



Instead of "How to decompile?" we ask
"Given a candidate decompilation, how good a match is it?"

Similar shift occurred for statistical machine translation

Take away slide

- Software RE is tedious, expertise required
- Code reuse is common in software, malware included
- We propose reframing: **software RE as a search problem**, relying on existing and available software to obtain source code
- Q: How can we do this in a way that is compiler-agnostic? (Assuming we deal with packers, obfuscators)

How we achieve this

- Design trade-offs
- Feature extraction
- Indexing & Querying
- Experimental results

Design space

- We want features that can uniquely identify functions
- We want speed + accuracy: We chose **Speed** first
- Speed meant that we chose static over dynamic analysis (Assumption: no obfuscation)
- We studied heuristic features from existing literature that can be extracted directly from a disassembly:
  - Instruction mnemonics n-grams, n-perms
  - Control-flow sub-graphs, extended sub-graphs
  - Data constants

Instruction mnemonics

- Instruction mnemonic (textual) differs from an opcode (hex), e.g. $0x8b$ (load) and $0x89$ (store) map to '*mov*'
- Assume a Markov property, n^{th} token is influenced by the previous n - 1 tokens
- Considered $n = 1, 2, 3, 4$

$$push, mov, push \longrightarrow 0x73f973 \longrightarrow \text{XvxFGF}$$

n-grams vs n-perms

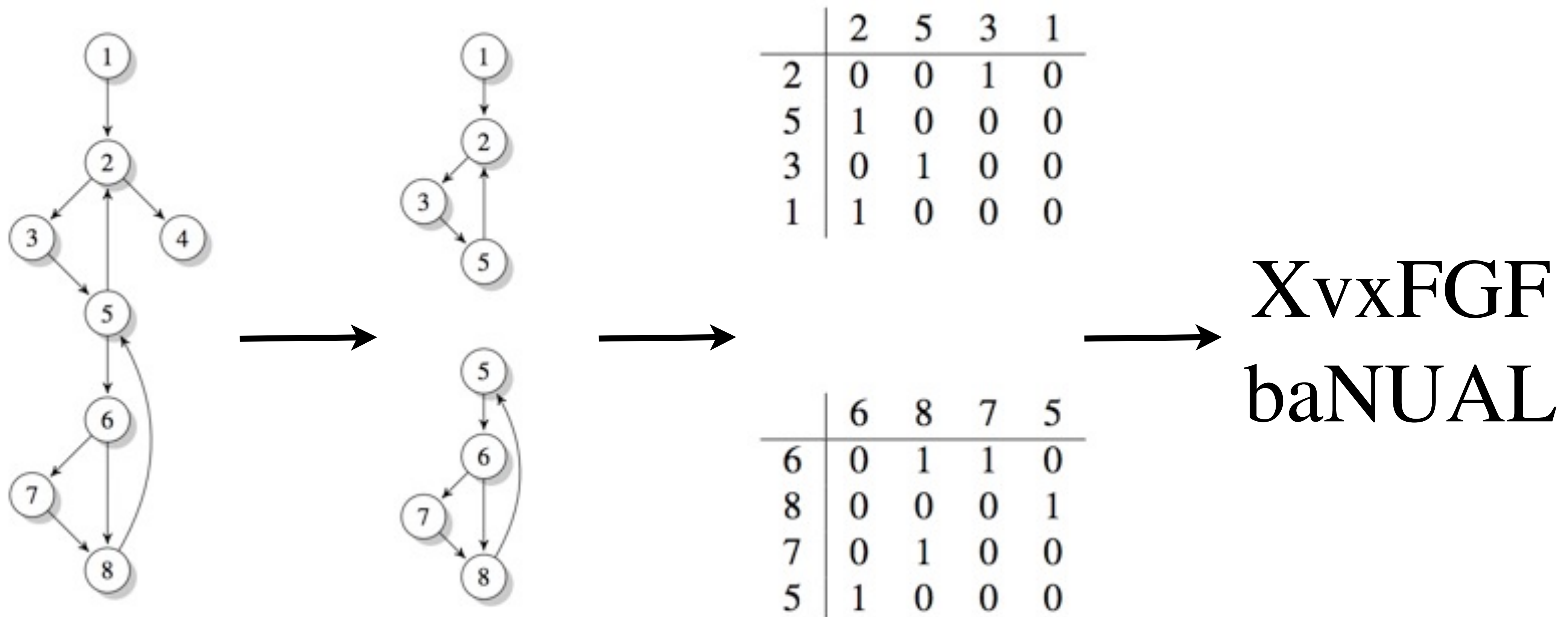
- n-gram is sequence-based, n-perm is set-based

```
mov  ebp, esp                 mov  ebp, esp
sub  esp, 0x10                movl -0x4(ebp), 0x1
movl -0x4(ebp), 0x1           sub  esp, 0x10
```

- For instance, two n-grams (mov, sub, movl) & (mov, movl, sub)
- Only 1 n-perm (mov, movl, sub)

Control-flow k-graphs

- k-graph is a connected sub-graph comprising k nodes, compute them all ($k = 3, 4, 5, 6, 7$)
- Convert to k-by-k matrix and compute its canonical form, rep as string (Nauty graph library)



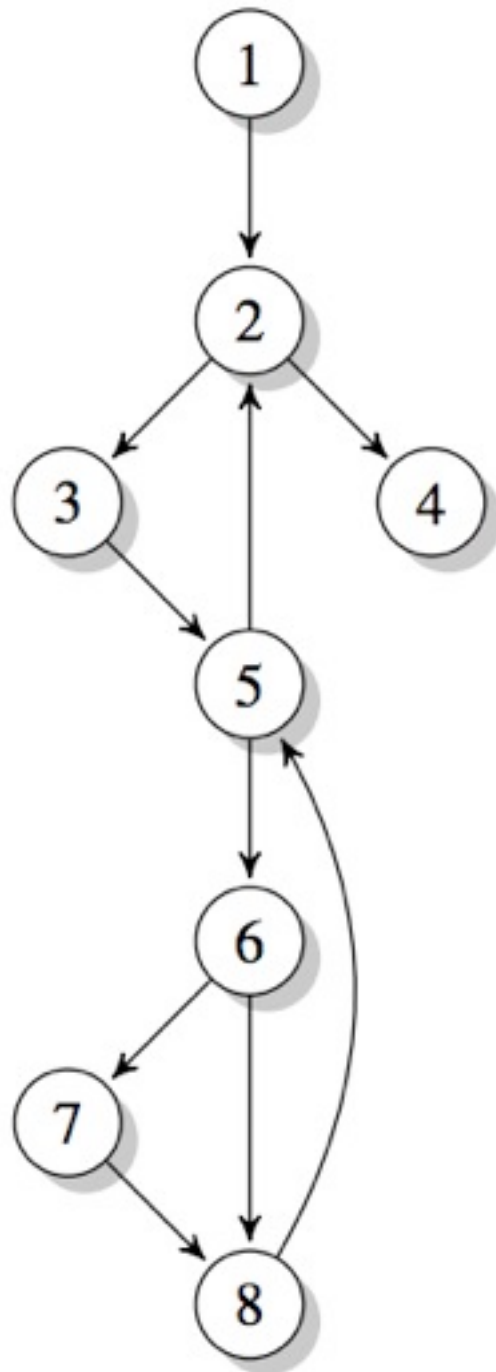
	2	5	3	1
2	0	0	1	0
5	1	0	0	0
3	0	1	0	0
1	1	0	0	0

	6	8	7	5
6	0	1	1	0
8	0	0	0	1
7	0	1	0	0
5	1	0	0	0

Extended *k*-graphs

- One shortcoming of *k*-graphs: uniqueness low for small *k*
- We propose extended *k*-graphs
- Extended *k*-graph includes edges that have one end point at an internal node, but have another at an external virtual node, $V*$

Extended *k*-graphs



	1	2	3
1	0	1	0
2	0	0	1
3	0	0	0

	3	5	6
3	0	1	0
5	0	0	1
6	0	0	0

k-graph

	1	2	3	V^*
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
V^*	0	1	0	0

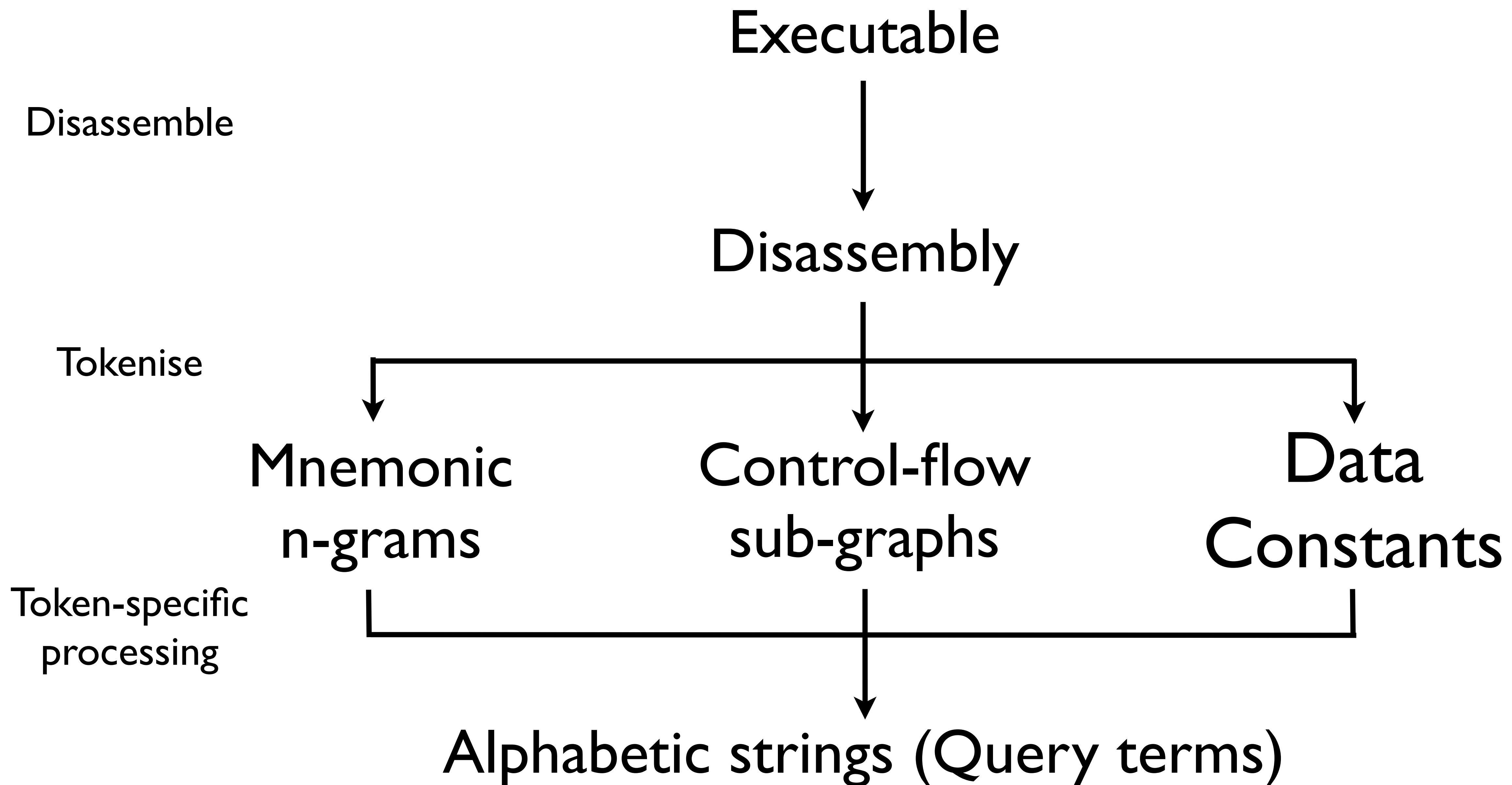
	3	5	6	V^*
3	0	1	0	0
5	0	0	1	0
6	0	0	0	1
V^*	1	1	0	0

Extended *k*-graph

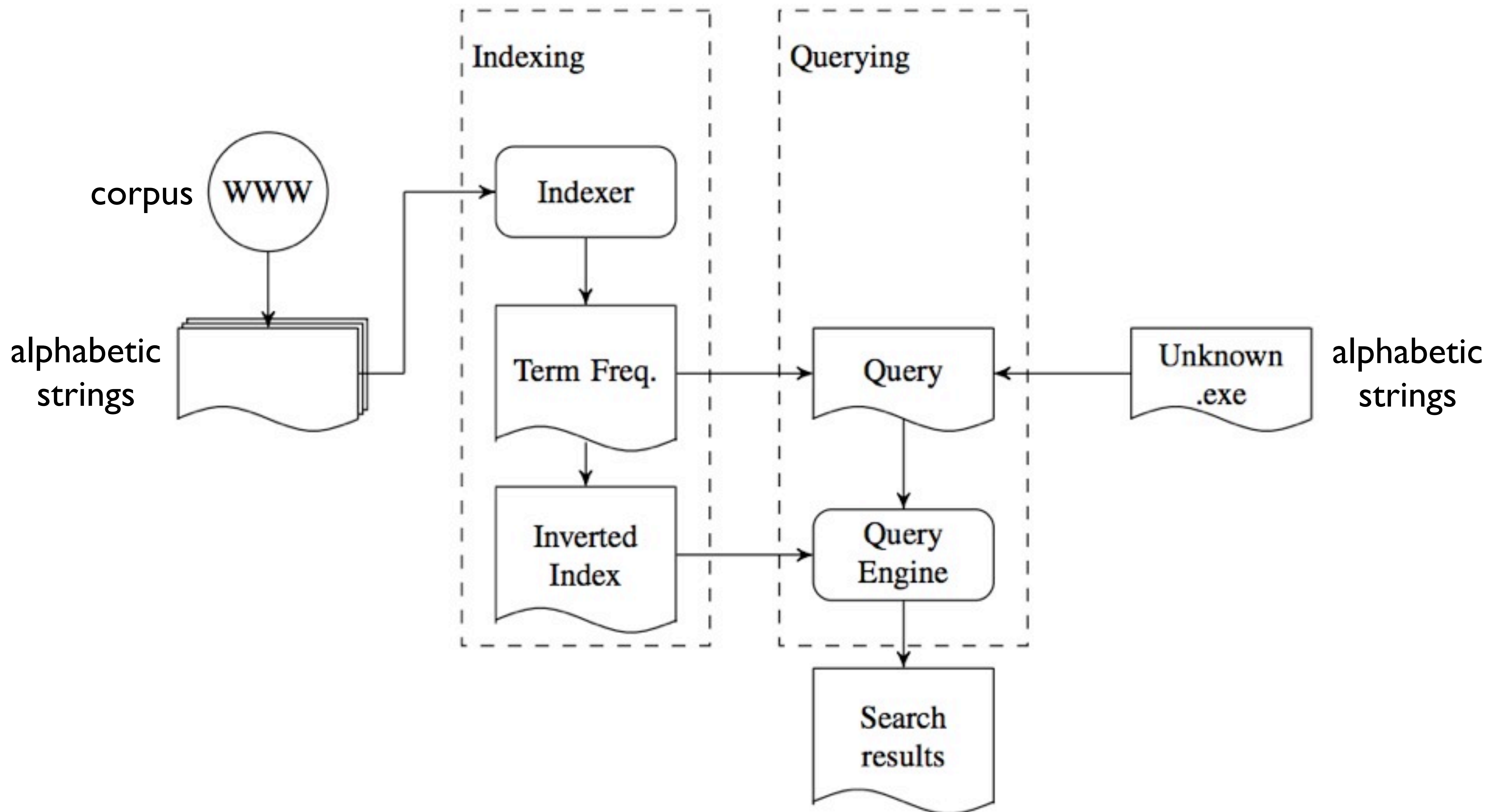
Constants

- Empirical observation that data constants do not change with compiler or options
- Considered 32-bit integers and strings
- Immediate operands, pointer offsets (excluding stack and frame pointer offsets)
- Integer may be an address, do a lookup

Feature extraction



Indexing & querying



Indexing & Querying

- 2 query models—the Boolean model (BM), and vector space model (VSM)
- BM is set-based, boolean operators such as AND, OR and NOT
- VSM is distance-based, weight vectors computed via normalised term frequencies
- Our model is based on the combination of the two: documents are first filtered via the BM, then ranked and scored by the VSM

Indexing & Querying

- Executable is abstracted to set of terms
- 3 strategies to deal with long queries, for desired query length l_Q
- Term de-duplication, Padding and Unique term selection
- Term de-duplication (up to 2 x l_Q) a simple strategy that reduces the query size
- Padding: Include common terms prepended by `NOT`
- E.g. For $l_Q = 3$, and query is `A AND B`, pad with `A AND B AND NOT C`
- Unique term selection: Select only the terms with frequency < $df_{threshold}$

Scoring

- Default CLucene scoring function

$$Score(Q, D) = coord(Q, D) \cdot C \cdot \frac{V(Q) \cdot V(D)}{|V(Q)|}$$

where *coord* is a score factor,

C is a normalisation factor,

V(Q)·V(D) is the dot product of the weighted vectors, and

|*V(Q)*| is the Euclidean norm

What makes a good model?

- True positives (tp): a correctly retrieved document relevant to the query
- False positive (fp): an incorrectly retrieved irrelevant document
- False negative (fn): a missing but relevant document

$$precision = \frac{tp}{tp + fp} \qquad recall = \frac{tp}{tp + fn}$$

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall} \qquad F_2 = \frac{5 \cdot (precision \cdot recall)}{(4 \cdot precision + recall)}$$

Implementation

- Disassembly: Dyninst binary instrumentation framework (http://dyninst.org)
- Indexing & Querying: CLucene text search engine (http://clucene.sourceforge.net)
- Term frequency map is a Bloom filter
- Code abstraction: 10,500 lines of C++
- Indexing/Querying: 1,000 lines of C++

Questions

- Optimal value of $df_{threshold}$?
- Accuracy of various abstractions?
- Accuracy for different compilers?
- Accuracy for different compiler options?
- Timing

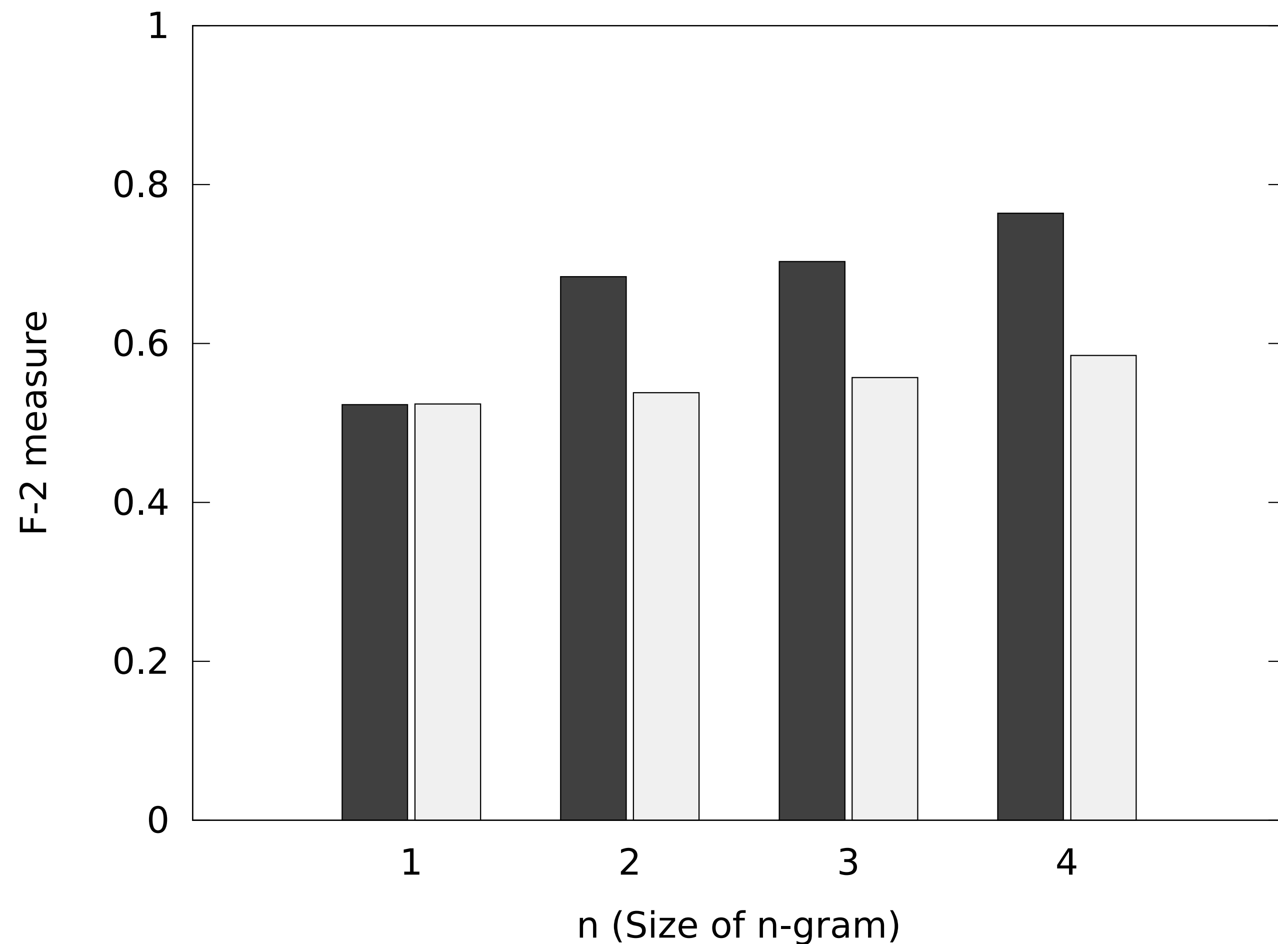
Datasets

- GNU C Library 2.16 (glibc)
  - 2,706 functions, 1.18 MLoC
  - Compiled with gcc -O1, -O2
- GNU coreutils 6.10 (coreutils)
  - 1,205 functions, 70,000 LoC
  - Compiled with gcc, clang

$df_{threshold}$

$\leq df_{threshold}$	Precision	Recall	F_2
1	0.202	0.395	0.331
2	0.177	0.587	0.401
3	0.165	0.649	0.410
4	0.161	0.677	0.413
5	0.157	0.673	0.406
6	0.160	0.702	0.418
7	0.159	0.709	0.419
8	0.157	0.708	0.415
9	0.157	0.716	0.418
10	0.155	0.712	0.414
11	0.151	0.696	0.405
12	0.152	0.702	0.408
13	0.153	0.705	0.410
∞	0.151	0.709	0.408

n-grams vs n-perms



n	n-gram	n-perm
1	121	121
2	1483	306
3	6337	542
4	16584	889

n-grams out-performed n-perms for $n > 1$

Possible explanation: Unique terms

k-graphs vs ex. k-graphs

	glibc					
	k-graph			extended k-graph		
	Precision	Recall	F_2	Precision	Recall	F_2
3-graph	0.070	0.133	0.113	0.022	0.062	0.046
4-graph	0.436	0.652	0.593	0.231	0.398	0.348
5-graph	0.730	0.700	**0.706**	0.621	0.600	0.604
6-graph	0.732	0.620	0.639	0.682	0.622	**0.633**
7-graph	0.767	0.609	0.635	0.728	0.610	0.631
	coreutils					
	k-graph			extended k-graph		
	Precision	Recall	F_2	Precision	Recall	F_2
3-graph	0.110	0.200	0.172	0.042	0.080	0.068
4-graph	0.401	0.586	0.537	0.218	0.360	0.318
5-graph	0.643	0.623	**0.627**	0.553	0.531	0.535
6-graph	0.617	0.527	0.543	0.660	0.602	**0.613**
7-graph	0.664	0.560	0.578	0.663	0.566	0.583

Mixed n-grams

	glibc	coreutils
1+2-gram	0.682	0.619
1+3-gram	0.741	0.649
1+4-gram	**0.777**	**0.671**
2+3-gram	0.737	0.655
2+4-gram	**0.777**	**0.675**
3+4-gram	0.765	0.671

- 1+4-grams & 2+4-grams were best performers
- Out-performed the best n-gram model (coreutils: 0.764, glibc: 0.664)

Mixed k-graphs

	glibc F_2	*coreutils* F_2
3+4-graphs	0.607	0.509
3+5-graphs	0.720	0.630
3+6-graphs	0.661	0.568
3+7-graphs	0.655	0.559
4+5-graphs	0.740	0.624
4+6-graphs	0.741	0.624
4+7-graphs	0.749	0.649
5+6-graphs	0.752	0.650
5+7-graphs	**0.768**	**0.657**
6+7-graphs	0.720	0.624

5+7-graphs best performer for both sets

Constants

	Precision	Recall	F_2
glibc	0.690	0.679	0.681
coreutils	0.867	0.751	0.772

Possible explanation: None of the functions in *glibc* had strings, whilst 889 functions, or 40.3% of functions in *coreutils* did

Composite models

		Precision	Recall	F_2
glibc	4-gram/5-graph/constants	0.870	0.866	**0.867**
	1-gram/4-gram/5-graph/ 7-graph/constants	0.850	0.841	0.843
	4-gram/5-graph/constants ($r = 10$)	0.118	**0.925**	0.390
coreutils	4-gram/5-graph/constants	0.835	0.829	**0.830**
	2-gram/4-gram/5-graph/ 7-graph/constants	0.833	0.798	0.805
	4-gram/5-graph/constants ($r = 10$)	0.203	**0.878**	0.527

- More components not necessarily better
- Looked at recall rates for top 10 results

Results at a glance

Model	*glibc* F_2	*coreutils* F_2
Best n-gram (4-gram)	0.764	0.665
Best k-graph (5-graph)	0.706	0.627
Constants	0.681	0.772
Best mixed n-gram (1+4-gram)	0.777	0.671
Best mixed k-graph (5+7-graph)	0.768	0.657
Best composite (4-gram/5-graph/constants)	0.867	0.830

False negatives

- 342 from glibc: 206 had 6 instructions or less
- getfsent
  - In-lining of fstab_convert
  - Instruction substitution: xor ax, ax to mov ax, 0; call/leave/ret to leave/jmp
  - Instruction re-ordering
  - No n-grams, k-graphs, constants in common

Timing

		Average (s)	Worst (s)
Abstraction	n-gram	46.684	51.881
	k-graph	110.874	114.922
	constants	627.656	680.148
	null	11.013	15.135
Query construction		6.133	16.125
Query		116.101	118.005
Total (2410 functions)		907.448	981.081
Total per function		0.377	0.407

- Timing for 2,410 coreutils functions
- 0.407s per function in worst case
- Constants extraction can be streamlined further

Conclusion

- Software RE is tedious, expertise required
- Code reuse is common in software
- We propose reframing: **software RE as a search problem**
- Able to achieve F_2 rates of 0.867 & 0.830 combining mnemonics, k-graphs and constants

http://www.rendezvousalpha.com