



File Fragment Encoding Classification: An Empirical Approach

By

Vassil Roussev and Candice Quates

Presented At

The Digital Forensic Research Conference

DFRWS 2013 USA Monterey, CA (Aug 4th - 7th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment. As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

<http://dfrws.org>



dfrows 2013 • aug 4-7 monterey ca

file fragment encoding classification an empirical approach



vassil roussev

vassil@roussev.net

candice quates

candice@egobsd.org

main points

- fix the problem definition
 - : prior efforts use an unworkable approach
- empirical study of deflate data
 - : find ways to classify encoded content
- zsniff
 - : PoC tool utilizing our findings
- msx-13
 - : new reference data set for docx/xlsx/pptx

revisiting *file type* definition

- OS: an association between file extension/
magic number and an application
: operations: *new, open, print, ...*
- almost all prior work (implicitly) assumes
OS-defined types == distinct data encoding
- what could possibly go wrong?

simple embedding

- compound file types may embed differently encoded data:

container file type (*msx-13, 20k files*)

embedded encodings

	docx	xlsx	pptx
jpeg			
count	5,644	2,838	59,067
avg size (KB)	142	68	121
total size (MB)	802	193	7,147
percent of total size	40	10	36
png			
count	6,777	1,728	65,692
avg size (KB)	68	46	134
total size (MB)	462	80	8,820
percent of total size	23	4	44
gif/tiff			
count	574	193	4,261
avg size (KB)	102	32	160
total size (MB)	59	6	680
percent of total size	2.9	0.3	3

recursive file embedding

- compound file types may include recursive file/object embedding:

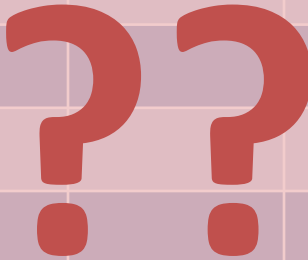
container file type (*msx-13, 20k files*)

**embedded
objects**

	docx	xlsx	pptx	total
bin (ole)	844	105	6,842	7,791
doc	106	30	221	357
docx	18	30	163	211
ppt			26	26
pptx	2		4	6
xls	71	4	693	768
xlsx	275	1	2,951	3,227
other	7		53	60
total	1,323	170	10,953	12,446

the matrix of confusion

	doc	ppt	docx	pdf	jpeg	zip	...
doc							
docx							
pptx							
pdf							
jpeg							
zip							
...							



→ this type of evaluation is largely meaningless

data encodings and file types

- (primitive) data encoding
 - : a set of rules for mapping pieces of data to a sequences of bits
 - : primitive encoding → not possible to reduce the rule set and still produce meaningful data encodings
- file type
 - : a set of rules for utilizing (sets of) primitive data encodings to serialize digital artifacts

the right questions

1. what is the **primitive data encoding** of the fragment?
 - : text, markup, code, deflate, jpeg, etc.
2. does the encoding contain **recursive encodings**?
 - : e.g., *base64*-encoded *jpeg*
3. is the fragment **part of** a compound file structure?
 - : e.g., *jpeg* fragment inside *pdf* file

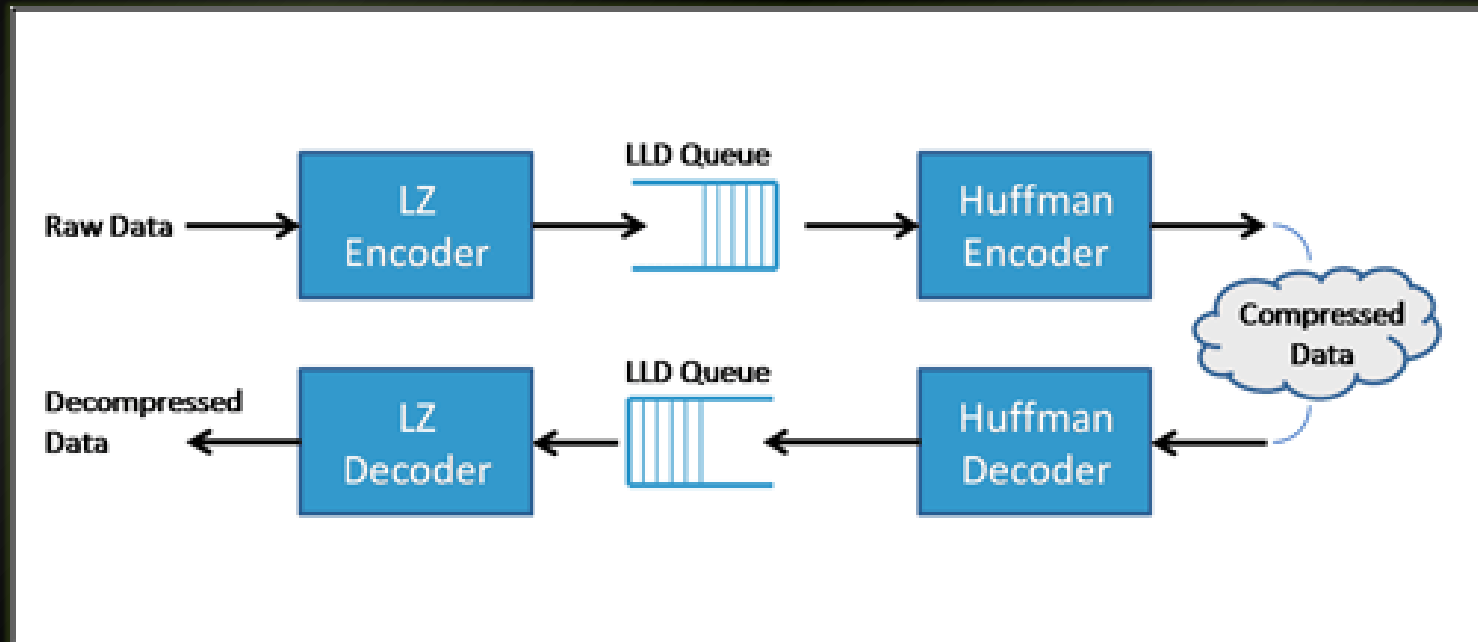
fragment content

- simple stuff
 - : text
 - : markup (html/xml/...)
 - : base16/32/64/85
 - : code
- compressed stuff
 - : text / markup
 - : images
 - : code
- encrypted stuff
 - : (compressed stuff that looks like nothing else)

the point of this work

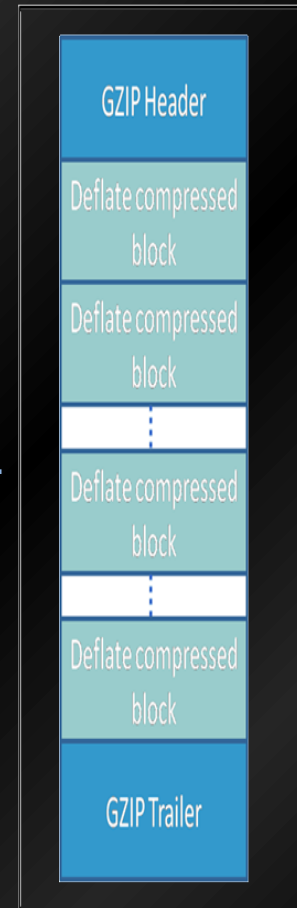
- classify compressed data (almost) as reliably as non-compressed data
- the problem
 - : once things get compressed, statistical features get obliterated
- the solution
 - : try to find compressed data headers and reason about the underlying data

DEFLATE compression

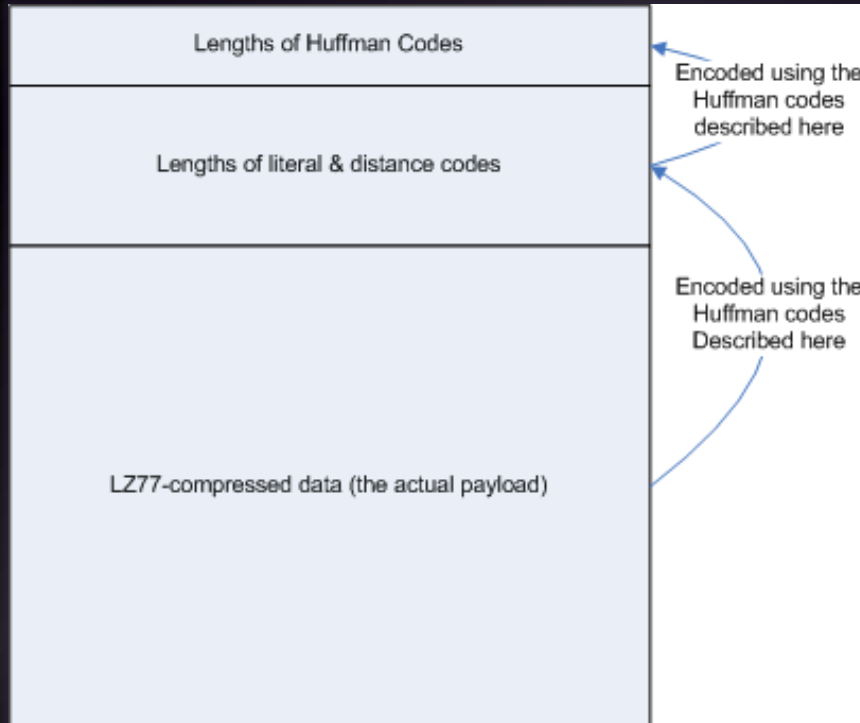


deflate stream format

- a deflate stream consists of a series of blocks.
- each block is preceded by a 3-bit header:
 - : 1 bit: Last-block-in-stream marker:
 - 1: this is the last block in the stream.
 - 0: there are more blocks to process after this one.
 - : 2 bits: Encoding method used for this block type:
 - 00: a stored/raw/literal section, between 0 and 65,535 bytes in length.
 - 01: a static Huffman compressed block, using a known Huffman tree.
 - 10: a compressed block complete with the Huffman table supplied.
 - 11: reserved, don't use.
- i.e., we have two bits to go on

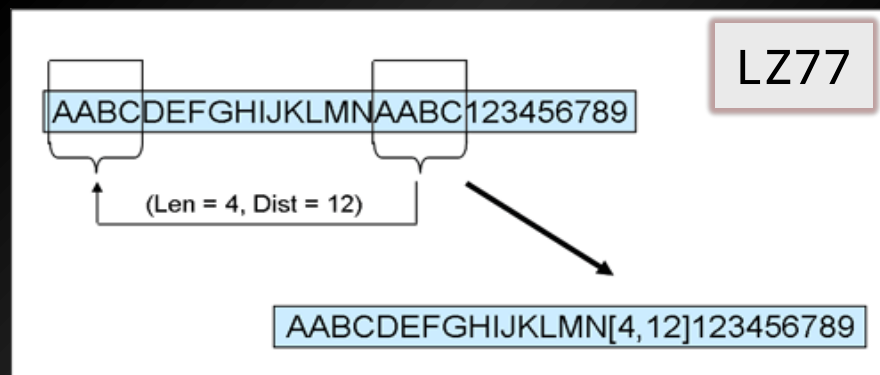
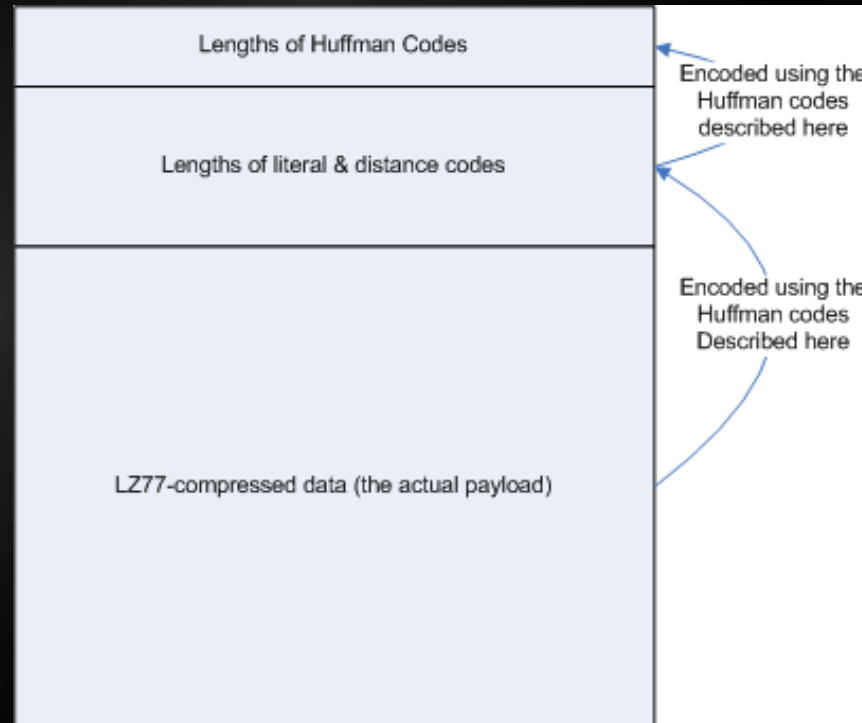


Huffman tree



- the Huffman tree is created w/ space for 288 symbols:
 - : 0–255: represent the literal bytes/symbols 0–255.
 - : 256: end of block – stop processing if last block, otherwise start processing next block.
 - : 257–285: combined with extra-bits, a match length of 3–258 bytes.
 - : 286, 287: not used, reserved and illegal but still part of the tree.

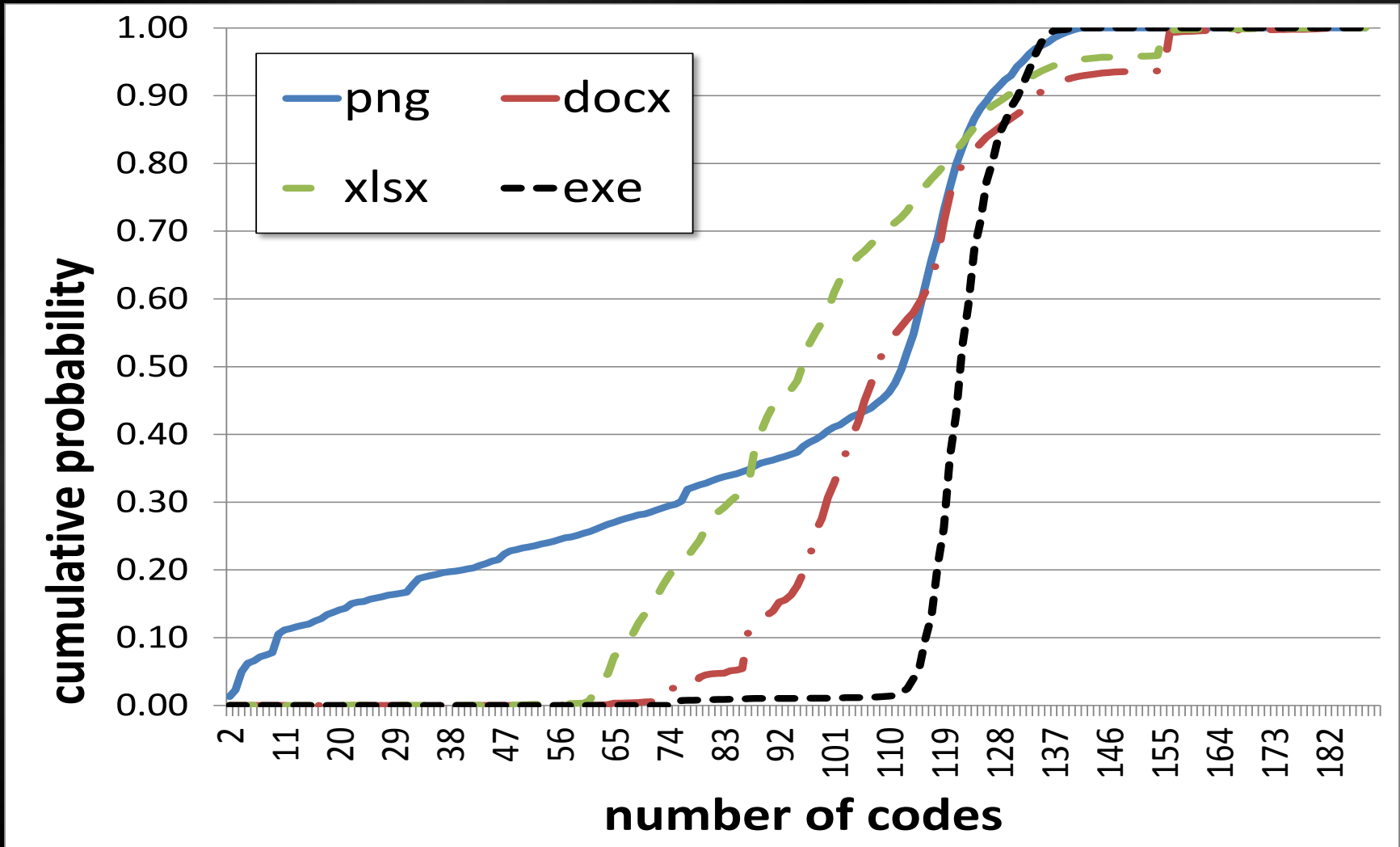
lz77



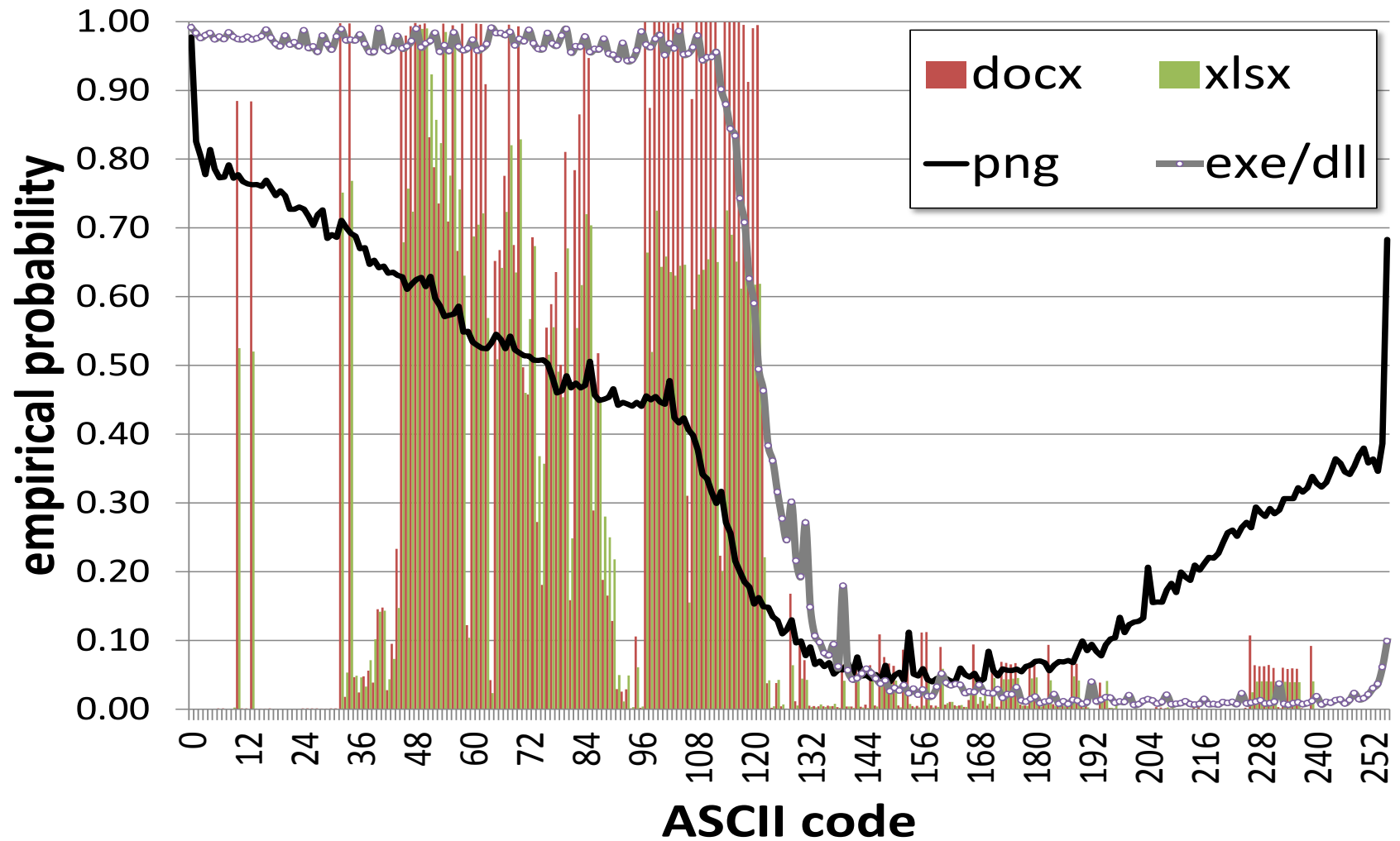
idea

- *maybe* different data uses different Huffman code books
- if so, we might be able to sniff the underlying data
- off to the charts
 - : w/ docx, xlsx, png, zlib-exe

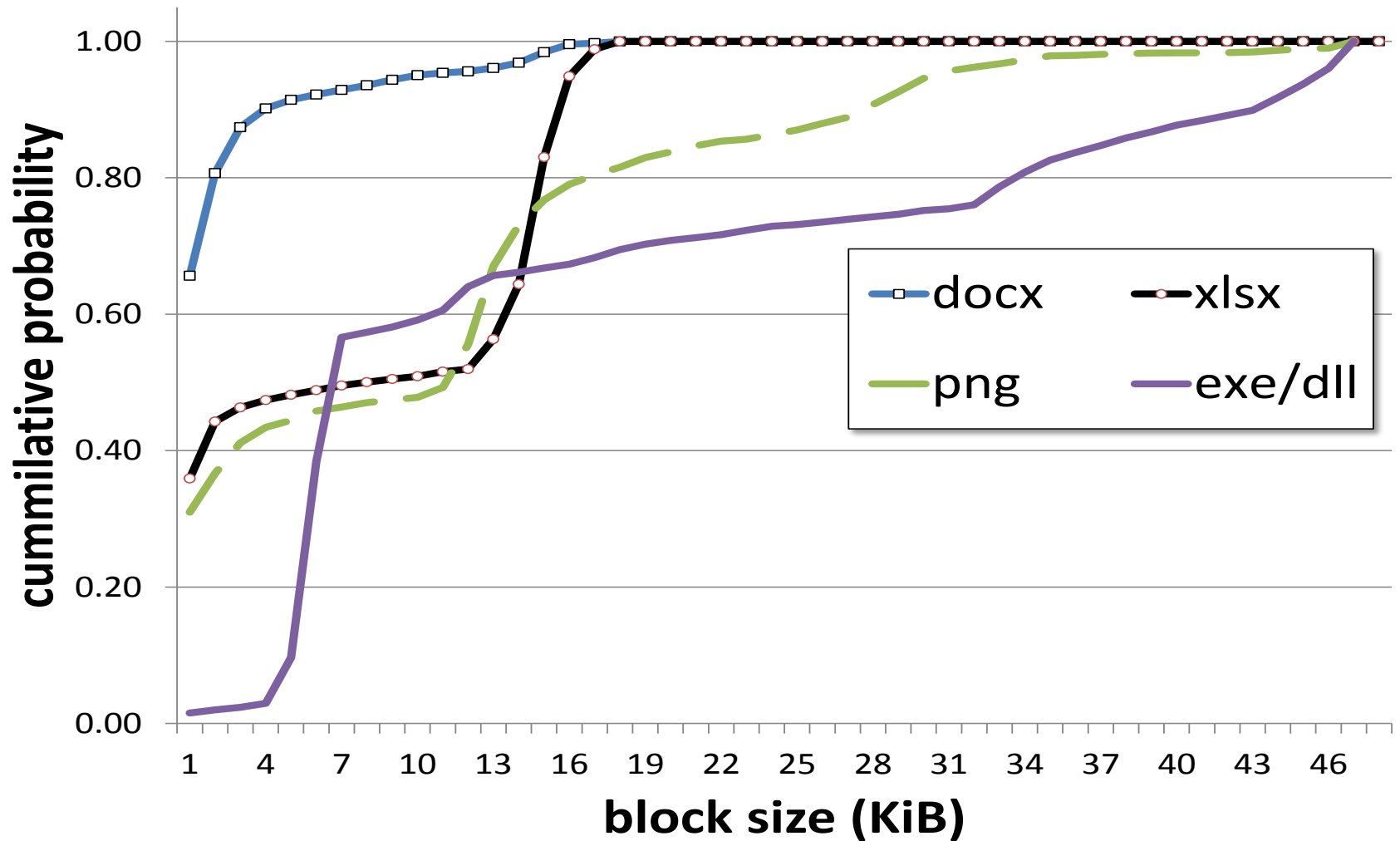
how many codes per block?



which codes?



how big should the fragment be?



quick *zsniff* demo

<http://github.com/zsniff/zsniff>

zsniff preliminary results

- as expected
 - : z-xml vs {png, z-exe} is easy
 - : png vs z-exe is much harder

	results	
data	z-xml	z-exe
z-xml	0.998	0.002
z-exe	0.003	0.997

	results	
data	png	z-exe
png	0.815	0.185
z-exe	0.062	0.938

msx-13

	docx	xlsx	pptx
file count	7,018	7,452	7,530
total size (MB)	2,014	1,976	20,037
avg size (KB)	287	265	2,661

- See: roussev.net/msx-13
 - : list of original URLs, download scripts
- For researchers
 - : contact us and we'll provide direct data download

contributions

- fixes the problem definition
 - : prior efforts used an unworkable approach
- empirical study of deflate data
 - : shows ways to analyze/classify *deflate* content
 - : studied the effect of fragment size on results
- zsniff
 - : PoC tool utilizing our findings (alpha stage)
 - : <http://github.com/zsniff/zsniff>
- msx-13
 - : new reference data set for docx/xlsx/pptx

thank you!

Q & A