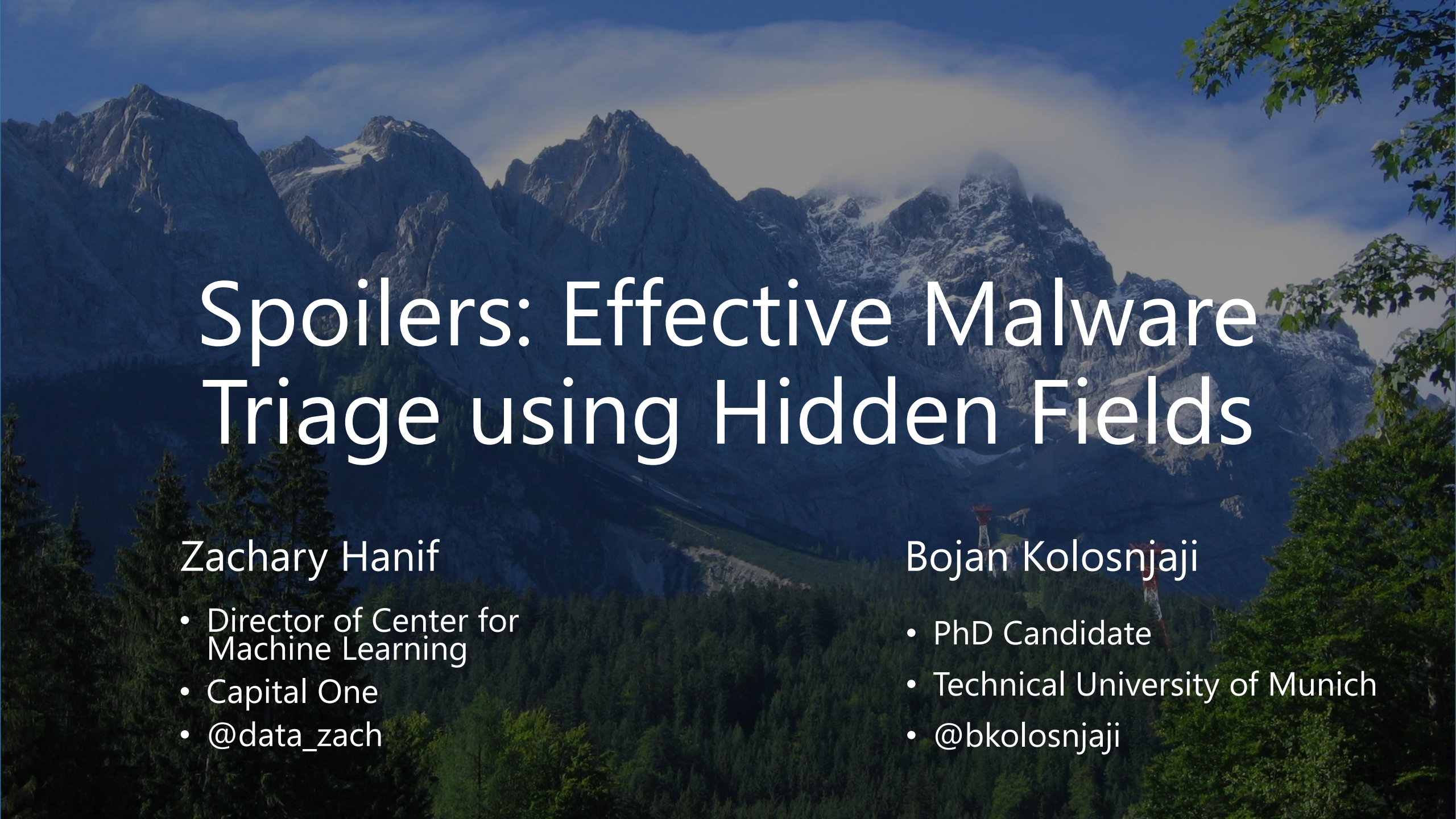


# Spoilers: Effective Malware Triage using Hidden Fields


Zachary Hanif

- Director of Center for Machine Learning
- Capital One
- @data_zach

Bojan Kolosnjaji

- PhD Candidate
- Technical University of Munich
- @bkolosnjaji

# 01: Why Are We Here?

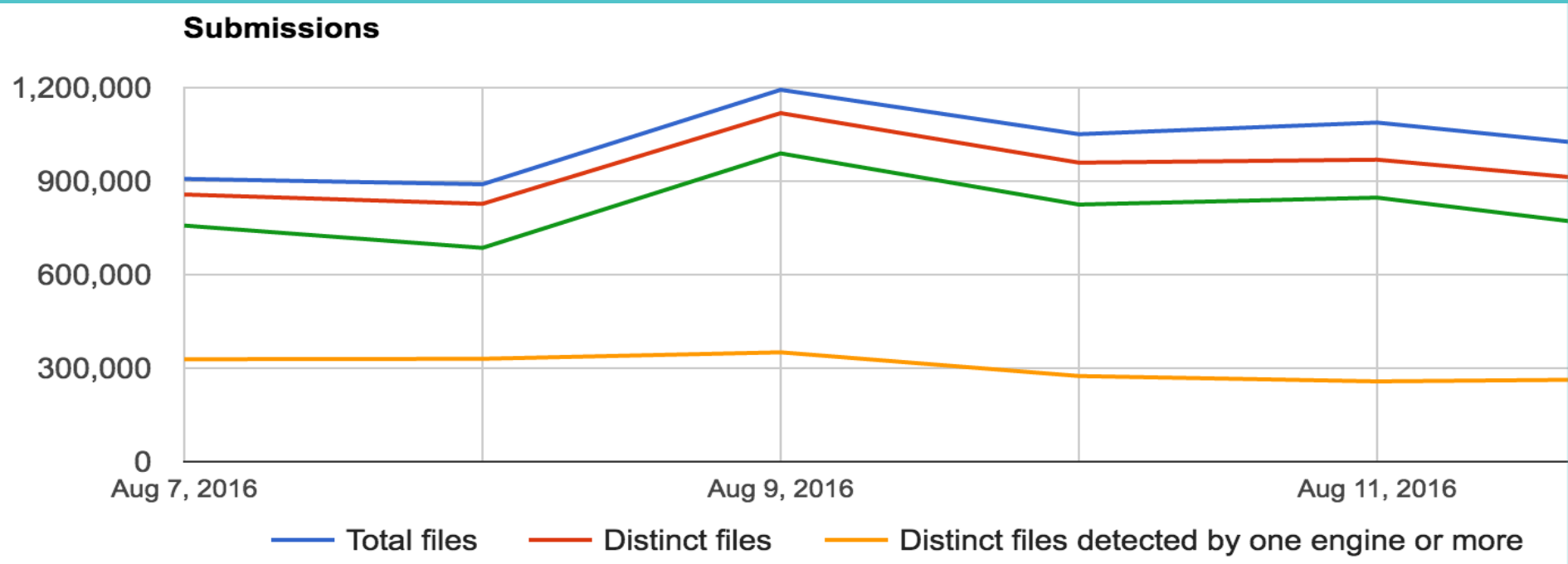
What is the problem?

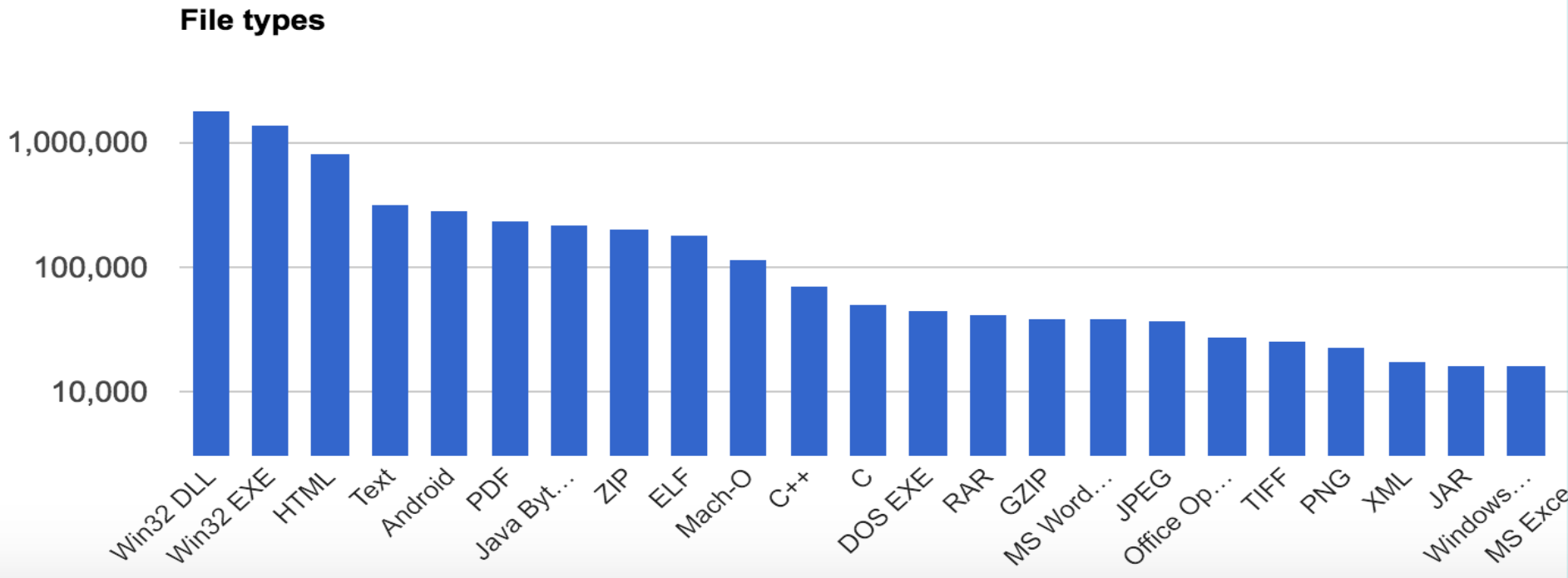
Why do you care?

# Data, Data, Everywhere

How do you:

- Triage?
- Find related data?
- Make sense of everything?



# What about this obscure PE32 field?

Overlooked, poorly documented, inaccurate assessments

# Rich Header

Rich Header is an overlooked and poorly understood aspect of PE32

- **Product ID:** Unique identifier for major and generic library packages
- **Version:** Reports version information of the product
- **Times Used:** Records the number of times the linker accesses the product

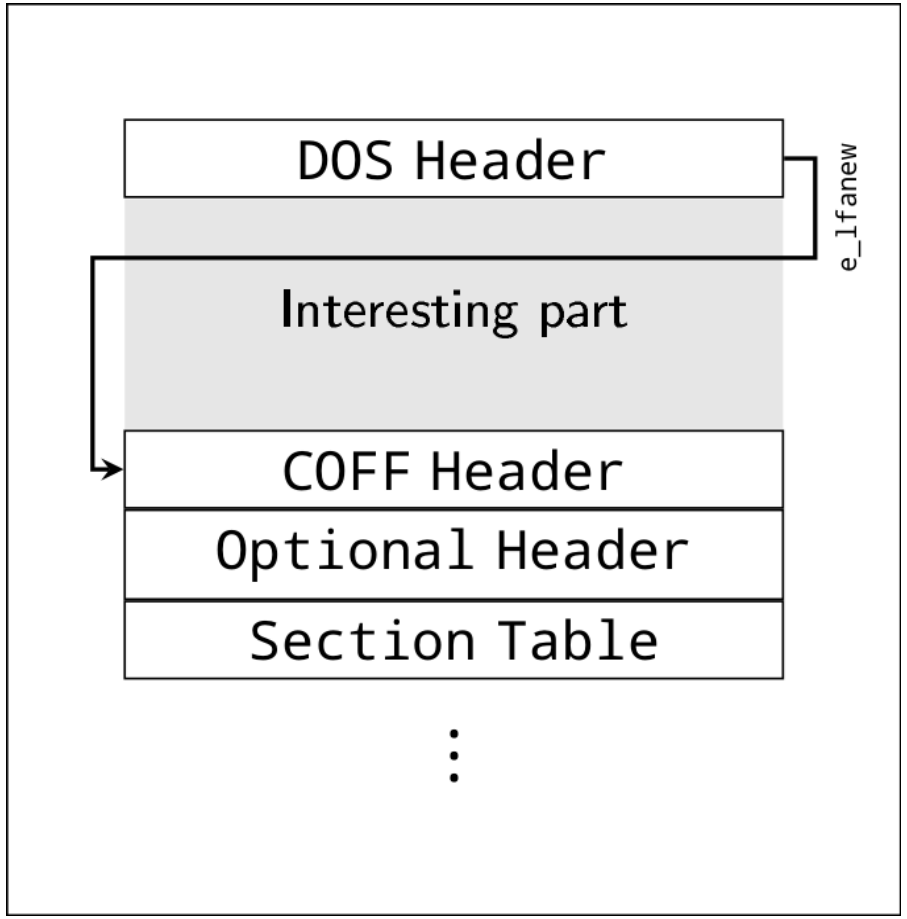
# 02: Background

PE32 File Format

Compiler Tool Chain

# PE32 File Format

**Stub** between DOS and COFF header:

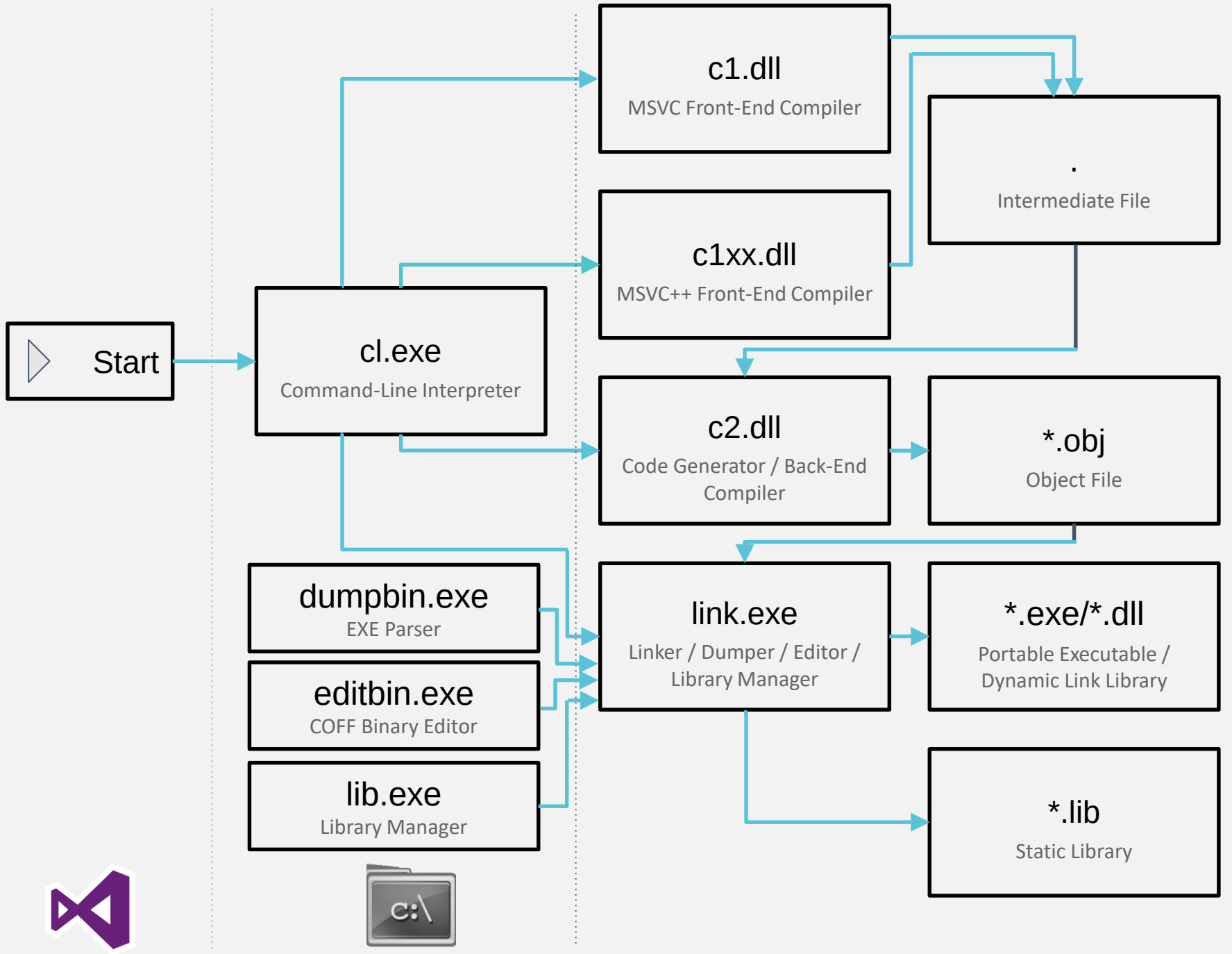



- **DOS program** printing "This program cannot be run in DOS mode"
  - Documented by Microsoft
  - Can be replaced by any valid MS-DOS application using MSVC's /STUB compiler flag
- **RICH header** containing unknown bytes terminated by the string "Rich" and a magic number
  - Never officially mentioned by Microsoft
  - No consistent explanation available

# MSVC Compiler Toolchain

Consisting of:

- Command-Line Interpreter
- C/C++ Frontend
- Code Generator
- (Multi Purpose) Linker



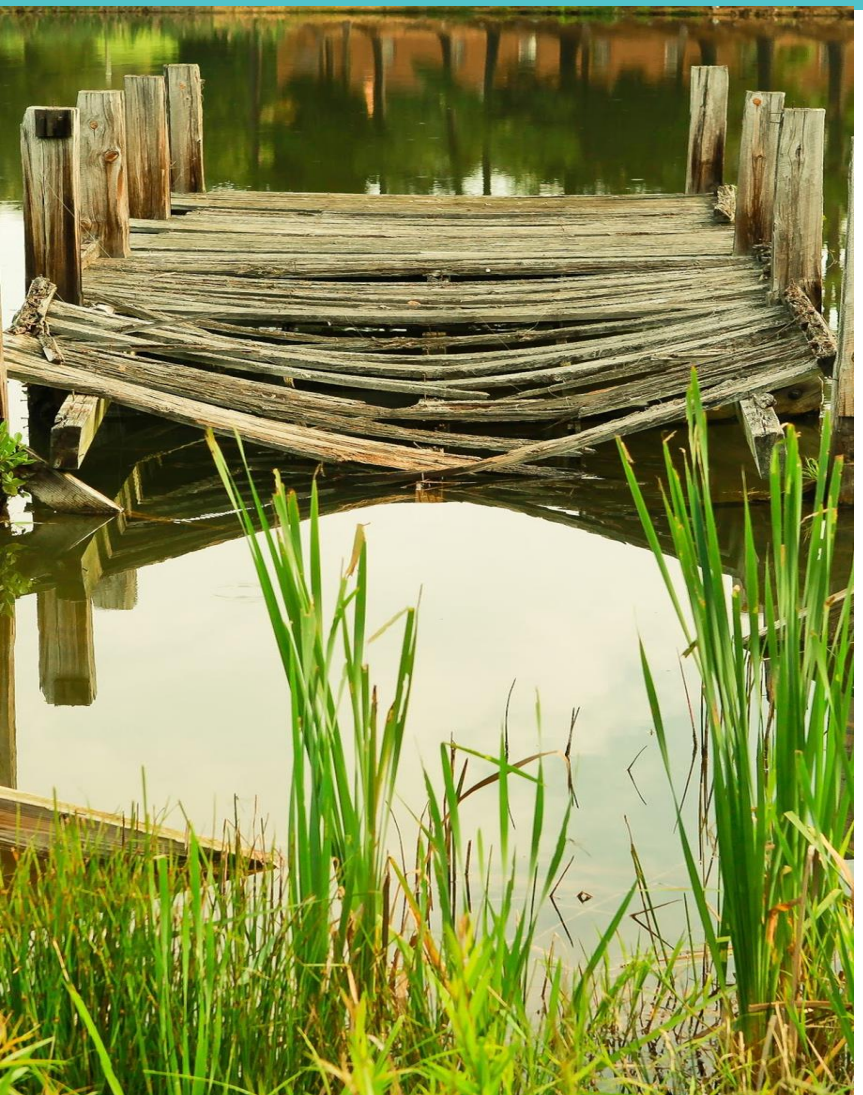
# 03: Rich Header

What does it contain?

How is it created?

How is it extracted?

# Rich Header: Obfuscated, Undocumented, Part of the PE Header



01

**History**

Included in MS Toolchain since Visual Studio 6 (1998) and maybe even earlier. First discussed in 2004 and reverse engineered in 2008 by Daniel Pistelli

02

**Maintenance**

Each iteration of the Microsoft Toolchain adjusts how the Rich Header is generated and updates product mapping

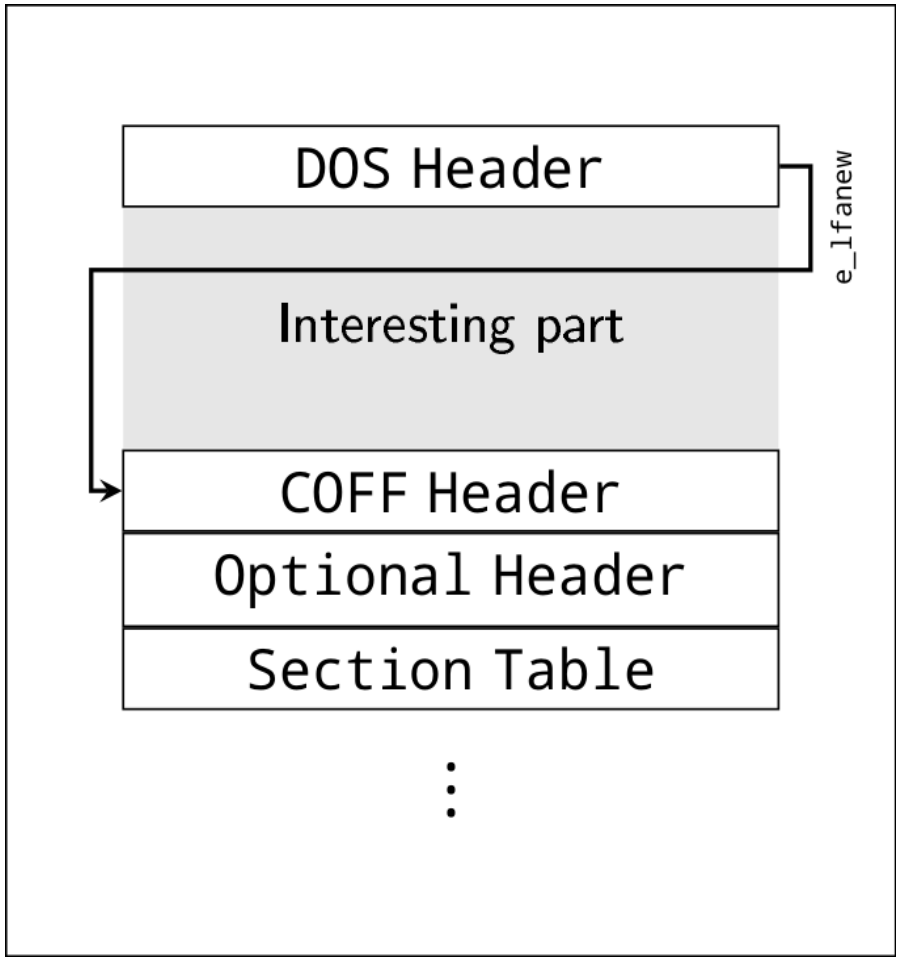
03

**Some Issues**

Most of the knowledge about the Rich Header is inaccurate or incomplete

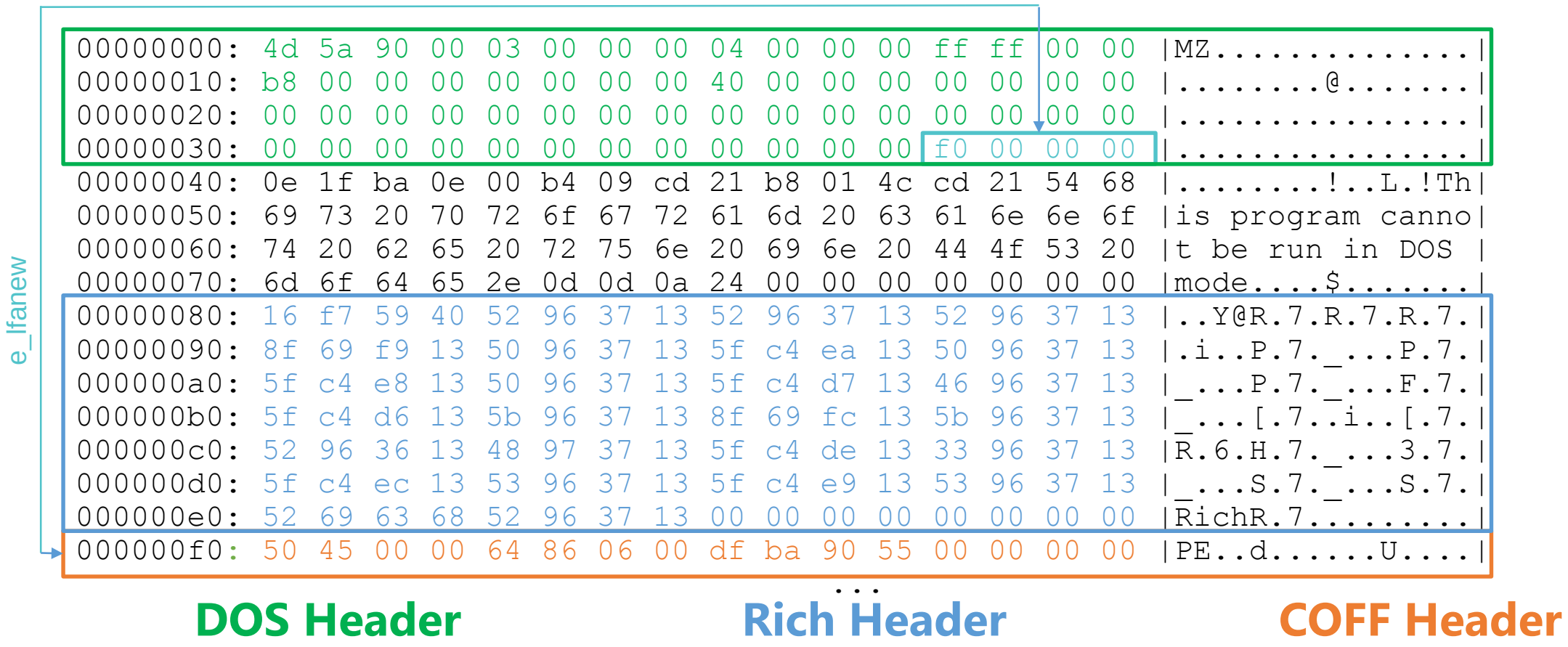
# PE32 File Format

**Stub** between DOS and COFF header:




- **DOS program** printing "This program cannot be run in DOS mode"
  - Documented by Microsoft
  - Can be replaced by any valid MS-DOS application using MSVC's /STUB compiler flag
- **RICH header** containing unknown bytes terminated by the string "Rich" and a magic number
  - Never officially mentioned by Microsoft
  - No consistent explanation available

# PE32 Header Structure

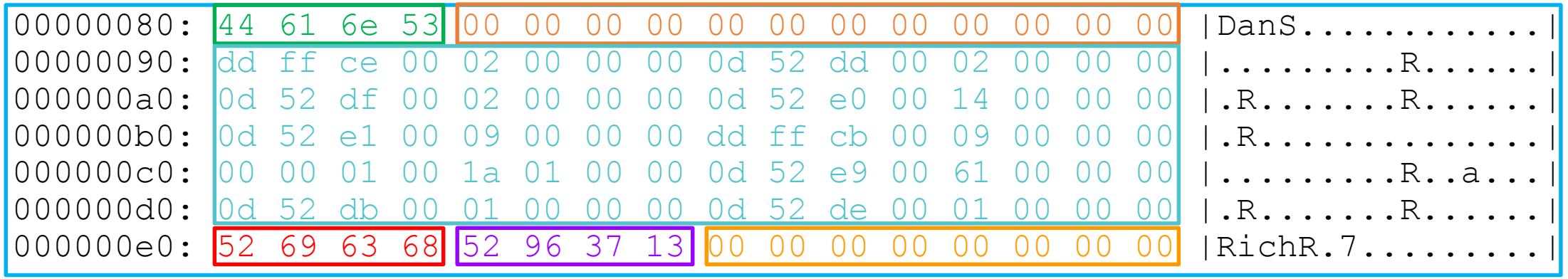


# Rich Header Structure

```
00000080: 16 f7 59 40 52 96 37 13 52 96 37 13 52 96 37 13 |..Y@R.7.R.7.R.7.|
00000090: 8f 69 f9 13 50 96 37 13 5f c4 ea 13 50 96 37 13 |.i..P.7._...P.7.|
000000a0: 5f c4 e8 13 50 96 37 13 5f c4 d7 13 46 96 37 13 |_...P.7._...F.7.|
000000b0: 5f c4 d6 13 5b 96 37 13 8f 69 fc 13 5b 96 37 13 |_...[.7..i..[.7.|
000000c0: 52 96 36 13 48 97 37 13 5f c4 de 13 33 96 37 13 |R.6.H.7._...3.7.|
000000d0: 5f c4 ec 13 53 96 37 13 5f c4 e9 13 53 96 37 13 |_...S.7._...S.7.|
000000e0: 52 69 63 68 52 96 37 13 00 00 00 00 00 00 00 00 |RichR.7.........|
```

- Footer (8 + x bytes)
  - "Rich" identifier
  - Checksum
  - Zero padding (*presumably* to next multiple of 16)

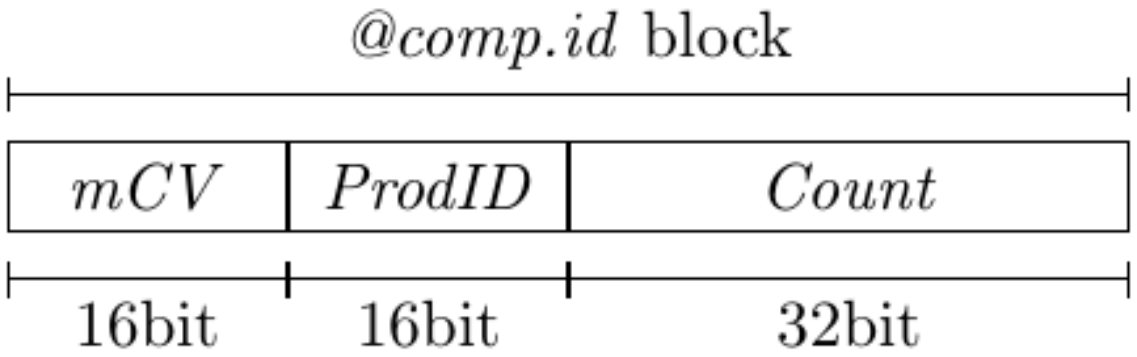
# Rich Header Structure




- Header (4 x 4 bytes)
  - "DanS"
  - Zero padding (fix!)
- @Comp.id Blocks (n x 8 bytes)
  - n @Comp.id Blocks
- Footer (8 + x bytes)
  - "Rich" identifier
  - Checksum
  - Zero padding (*presumably* to next multiple of 16)

} XORed with Checksum

# Structure of @comp.id




01

**mCV**

Minor version of the compiler used to make the product

02

**ProdID**

Unique identifier that specifies a specific identify or type of object…as well as compiler flags

03

**Count**

Specifies how often the specific ProdID and mCV were used by the linker

# ProdID

**Generic Identifier: Identifies the referenced object type and VS Release**

| ProdID | VS Release | Object Type | Generator |
|---|---|---|---|
| 0x105 | 2015 | C++ | c2.dll |
| 0x104 | 2015 | C | c2.dll |
| 0x103 | 2015 | Assembly | c2.dll |
| 0xff | 2015 | Resource File | cvtres.exe |
| 0xb4 | 2010 | C++ | c2.dll |
| 0x5e | .NET | Resource File | cvtres.exe |
| 0x15 | 6 | C | c2.dll |

**Unique Identifier: Appears to map to major libraries. Exact definition is unknown**

# Checksum

Rotate the DOS Heade bytes by their offset

Rotate contents of @comp.ids by their count

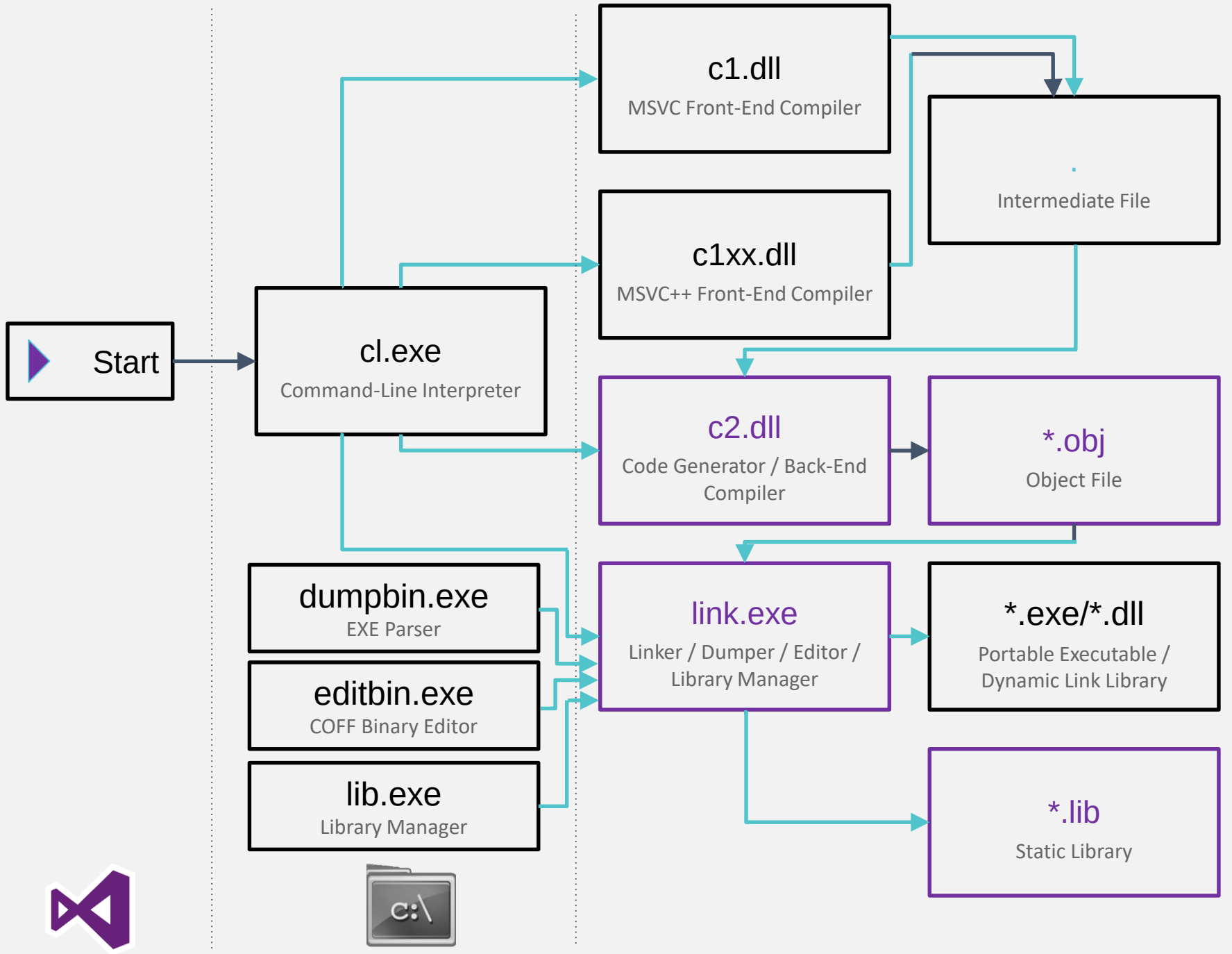
Only 37 of the 64 bits per @comp.id are checksummed!

```
## Rotate left helper function
rol32 = lambda v, n: \
        ((v << (n & 0x1f)) & 0xffffffff) | \
        (v >> (32 - (n & 0x1f)))

## raw_dat is a bytearray containing the exe's data
## compids is the list of deciphered @compid structs
## off is the offset to the start of the Rich Header
def calc_csum(raw_dat, compids, off):
    csum = off

    for i in range(off):
        ## skip e_lfanew as it's not initialized yet
        if i in range(0x3c, 0x40):
            continue
    csum += rol32(raw_dat[i], i)

    for c in compids:
        csum += rol32((c['prodid'] << 16) | c['mcv'], \
                c['count'])

    return csum & 0xffffffff
```

# Insertion of The Rich Header

Back-End Complier generate one @comp.id per object

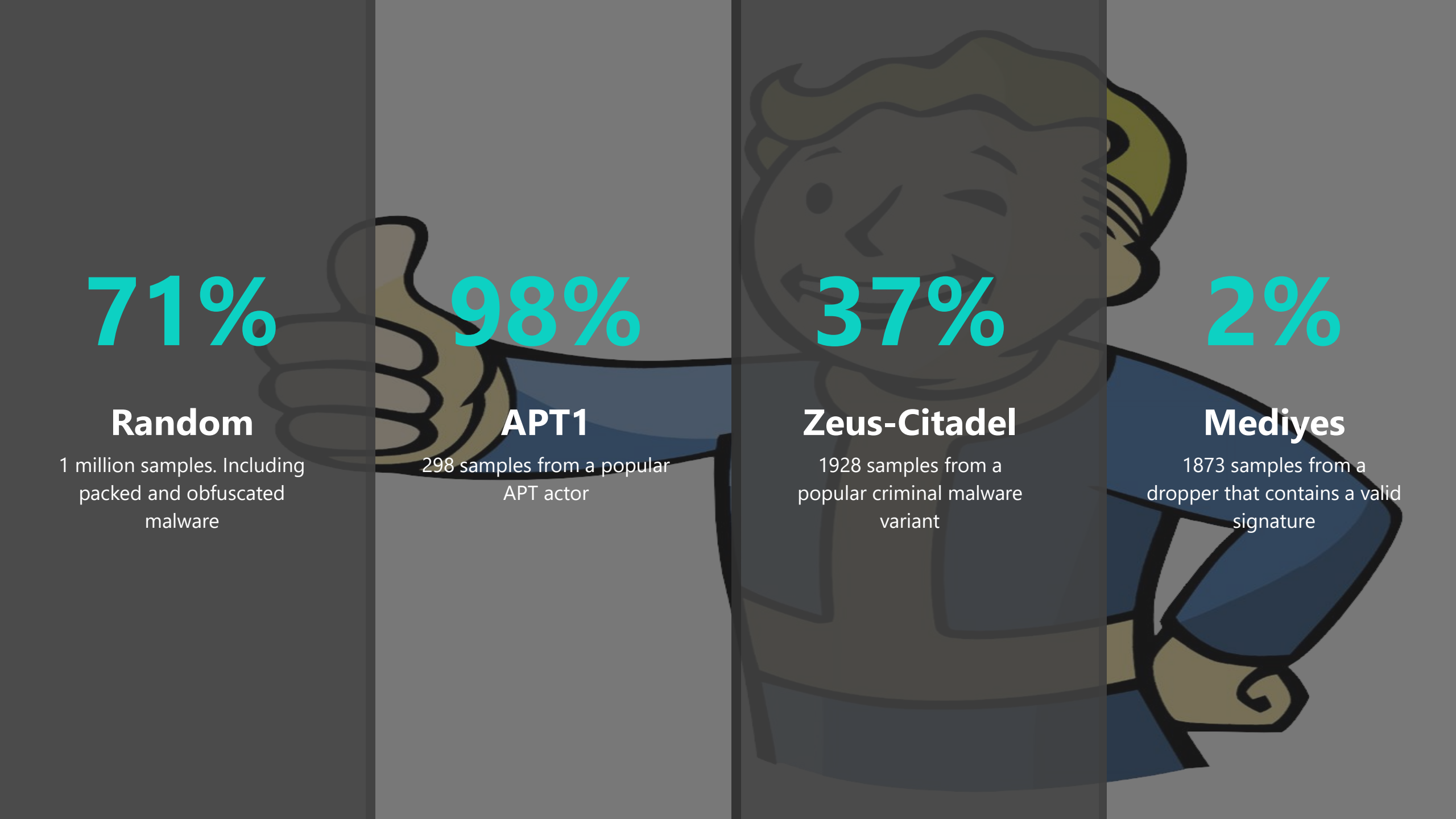
Linker collects @comp.ids from objects and puts them into the PE32



# 04: Statistics

Samples with a Rich Header

Samples without a Rich Header



**71%**

## Random

1 million samples. Including packed and obfuscated malware

**98%**

## APT1

298 samples from a popular APT actor

**37%**

## Zeus-Citadel

1928 samples from a popular criminal malware variant

**2%**

## Mediyes

1873 samples from a dropper that contains a valid signature

# The Microsoft Linker always adds the Rich Header

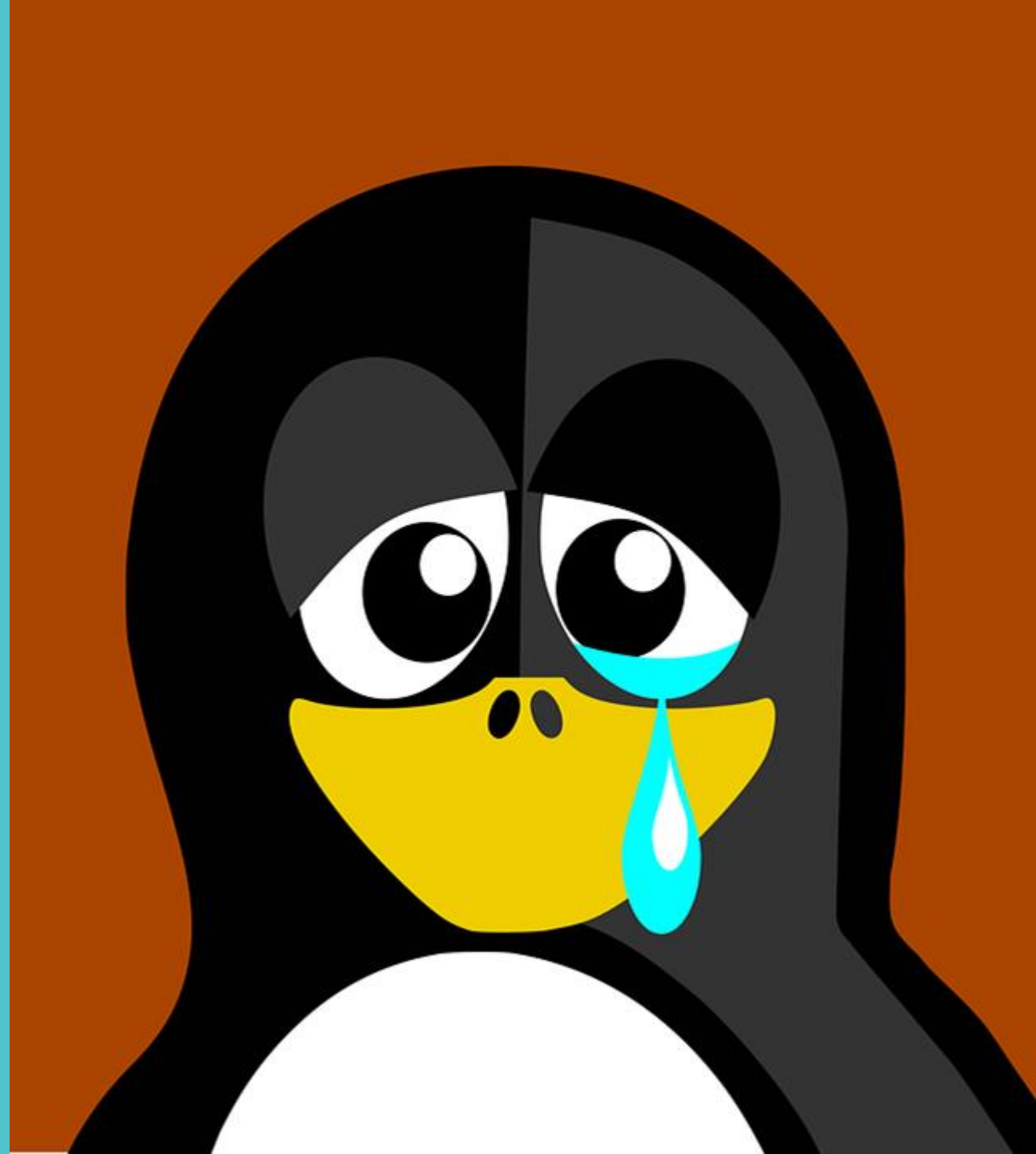
**No Header:**

- .Net
- MinGW
- GCC
- dUP

**With Header:**

- Visual Studios
- Intel
- UPX*
- ASPack*
- Nullsoft*

* More to come!



# Discrepancies are GREAT!

- [x] **Corrupt Checksum**

  Post modified binary

- [x] **Duplicate Entries**

  Packing Error

- [x] **Fast!**

  Very inexpensive check to perform. Out of 1 million samples, identified 22% were packed

# Rich Header Analysis

Assessments on what the binary does just based on their Rich Header


Can we do more?

# 05: Making Magic

Identify Suspicious Binaries

Similarity Matching

# With only the data in the Rich Header can we create the following:

**Fast**

Return the results in near real-time

**Similarity Matching**

Identify binaries that are similar. Potentially different versions or baked in

**Fingerprint Actors**

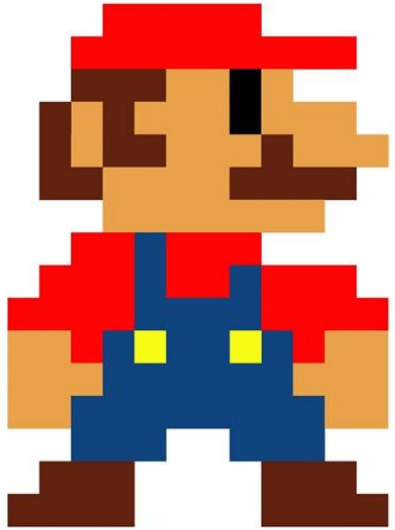
Identify binaries that were created under similar build environments

# Dimension Reduction

## Stacked Autoencoder

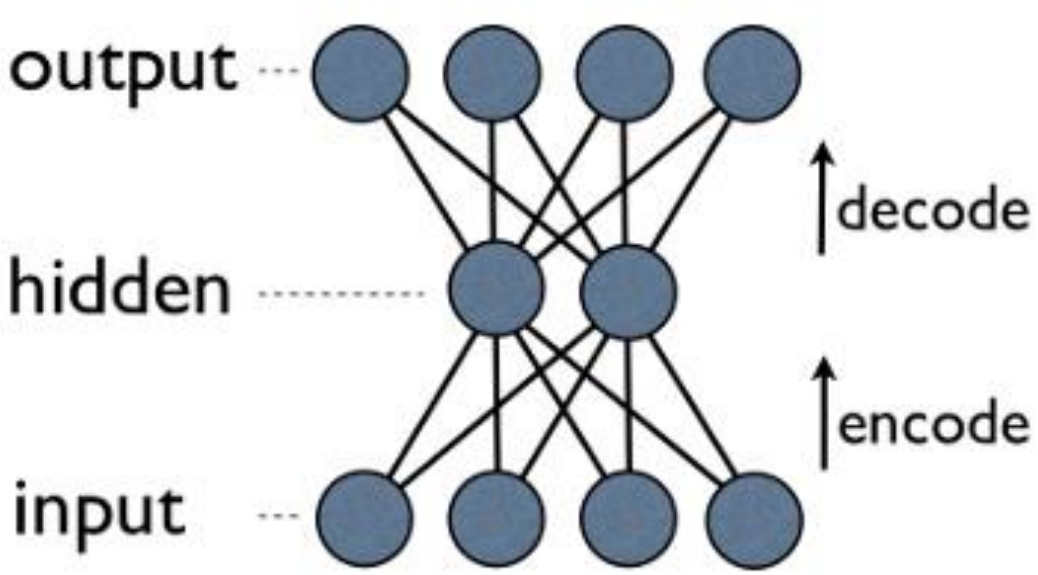
Benefits:

- Easier: denser lower-dimensional space
- Efficient: reduced memory requirement



VS.

# What Exactly IS an Auto-Encoder




- Neural network that attempts to create a **representation** of its input such that it can reconstruct that input as the output vector
- You **encode** your input to generate a comparable code, and you test if your code is a good representation of the original input by **decoding** it, and examining the difference
- Now, this sounds like we're just copying data, but by forcing the input through a hidden layer with an alternate dimensionality, we **generate vectors with useful properties**

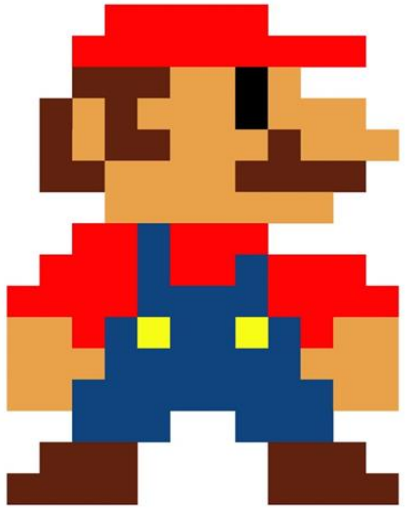
# Ok, That's Great, but What Use is That

- These codes are lower (in our case) dimensional representations of the original input
  - This means they are **smaller to represent** in memory, on disk, and they are less vulnerable to overfitting and misclassification due to the curse of dimensionality
  - The codes are directly **comparable within Euclidian space** – meaning that we can use them to find similar items
- Why not use PCA which is more commonly understood?
  - Autoencoders have **strong performance** when compared to PCA
  - Able to more closely represent **complex** or sparse data – which we have due to our one-hot encoding style
  - Autoencoders are fast to train, easy to create, and simple to validate

# Summary

- Autoencoder generated vectors
  - Are **smaller**, in memory and less CPU intensive during comparison
  - Are **less prone to overfitting** due to dimensionality reduction
  - Form a stronger **representation** of our data when compared to other dimensionality reduction methods
- We used a stacked autoencoder for this work
  - Performance better than single layer autoencoders in experiment
  - Reduced our feature vectors from 50 sparse dimensions to 10 dimensional dense vector
- This is all well and good, but KNN comparisons are O(n^2) – running this regularly is slow – to resolve this, we used a spatially indexed tree, a ball tree

# Didn’t get that?



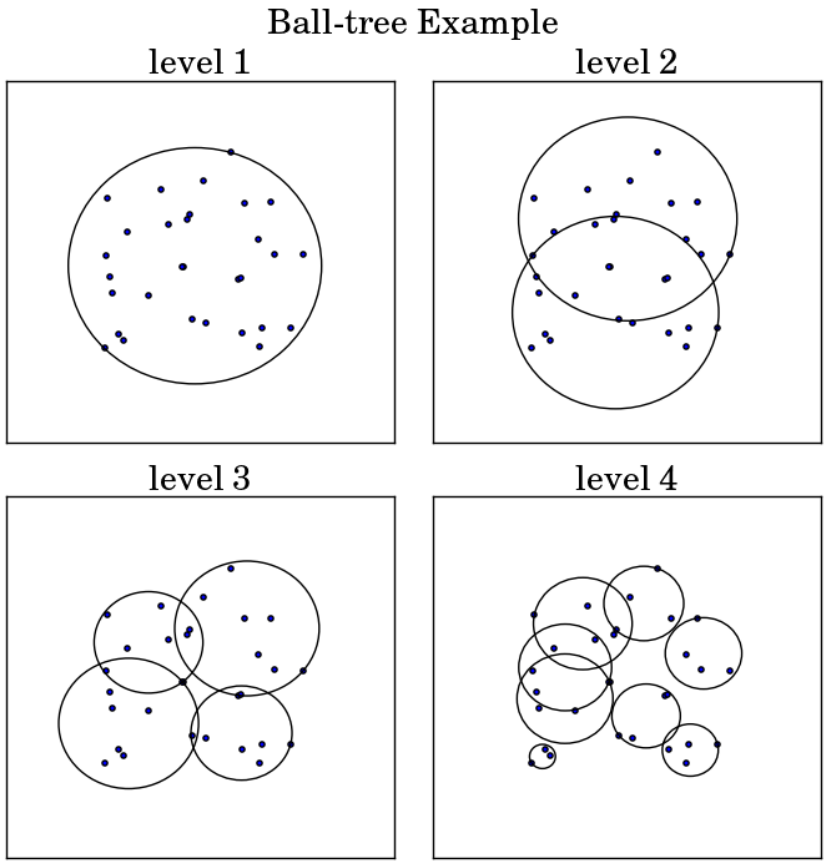
VS.

Benefits:

- Easier: denser lower-dimensional space
- Efficient: reduced memory requirement

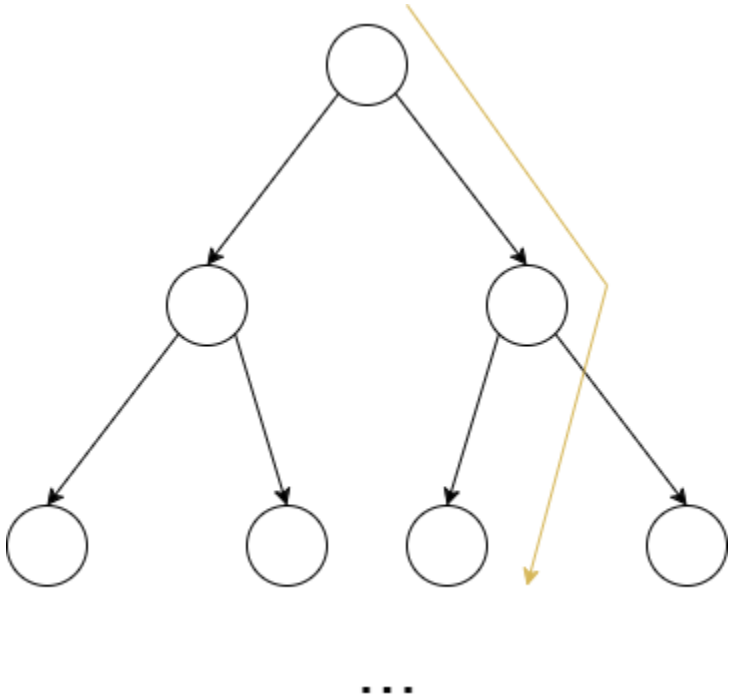
# Ball tree construction

- Ball trees are a kind of spatially partitioned tree data structure
  - Binary tree, every node contains a **ball** (hypersphere)
  - Each **internal node partitions data** points to it's **child spheres** based on the distance from the sphere's "center"
  - Each leaf node defines the smallest ball and enumerates all the data points within



# Ball Tree queries

- Time-memory tradeoff for KNN comparisons
  - Each new candidate for classification or neighbor identification is submitted to the tree, and the **nearest samples are returned**
  - Expected query time: O(log n)
  - Pre-built static tree effectively ensures balance



# Ball Trees in Practice

- This data structure brings queries down to sub-second speeds
  - Memory footprint <1GB for ~1,000,000 malware samples
- We were able to create ball trees on a per-dataset basis for inter-family and campaign examinations
  - This was a very **rapid process**, training was on the order of seconds, and ball tree creation was similar
  - This allowed us to do **similarity comparisons** across very large sample sets, with secondary comparisons against smaller, more focused sample families
    - Extremely fine-grained comparisons take approximately the same time as far larger ones

# Similarity Matching

## Nearest neighbours w/ Ball Tree

Benefits:

- Less pre-processing
- No predefined number of groups
- Fast lookups: 6.73ms Per 2 million

# 06: Case Studies

Identify Suspicious Binaries

Similarity Matching

# Case Study APT1

Based on SHA256:
F737829E9AD9A025945AD9C
E803641677AE0FE3ABF43B19
84A7C8AB994923178

All samples have different AV signatures

## Matching Rich Header

**Detected 1 sample**

**1:1 Match**
Identical functionality
Identical code base

Sha256 difference was from compiler artifacts

## Nearest Neighbors

**Detected 3 samples**

**1) Different Build Environment**
Library versions were slightly off

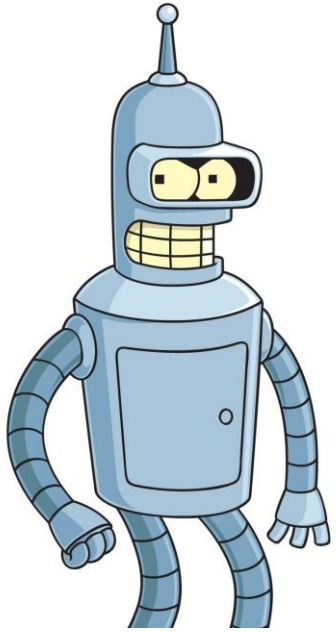
**2) Different Version**
Adds function “FlushFileBuffers”

**3) Version Upgrade**
Removes double write by calling strcat

# Case Study Zeus

Based on SHA256:
8471A205E1E85080B7230DB
19D773D43A559ECA7A4B892
E64E74C4E7E0A0D3BD

Most samples have a generic AV signature



## Matching Rich Header

**Detected 23 samples**

**1:1 Match**
Identical functionality

Assembly equivalent:
- XOR uses a “do while” versus “for” loop
- Code segments reordered

## Nearest Neighbors

**Detected 4 samples**

**Different version**
Identical functionality

XOR algorithm loops >8 times more

# Case Study Zeus Part 2

Based on SHA256:
7F1A07F484A8AE853DB9364
508A7BDFD3718BFA5E3115A
D941B216D0B662A880

Most samples have no signature or generic AV signatures

## Matching Rich Header

**Detected 36,606 samples**

**1:1 Match**
Identical functionality
A constant value changes

16,123 samples have no AV detection

## Nearest Neighbors

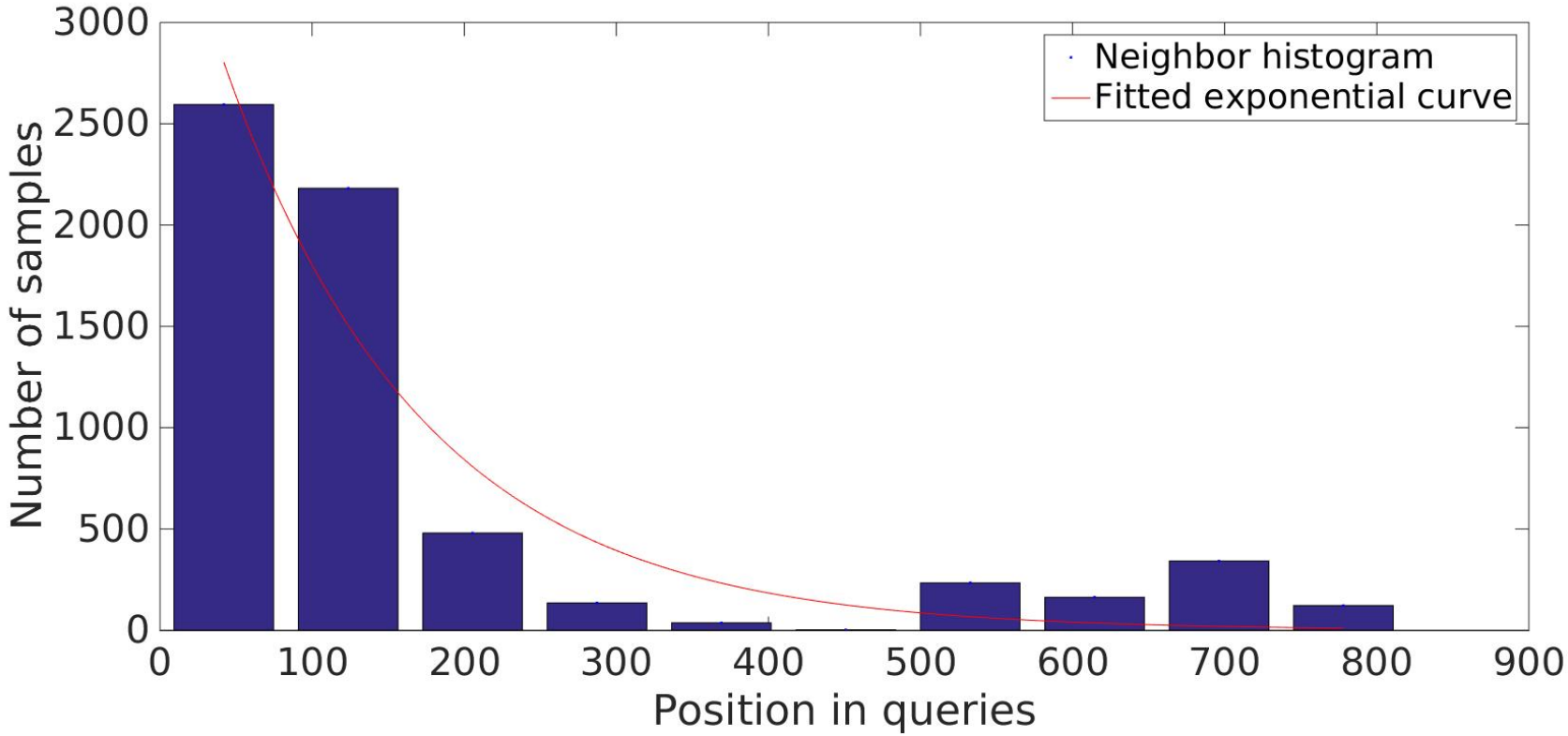
**Detected 1,567 samples**

**Different Build Environment**
Identical functionality
Library versions were slightly off

511 samples have no AV detection

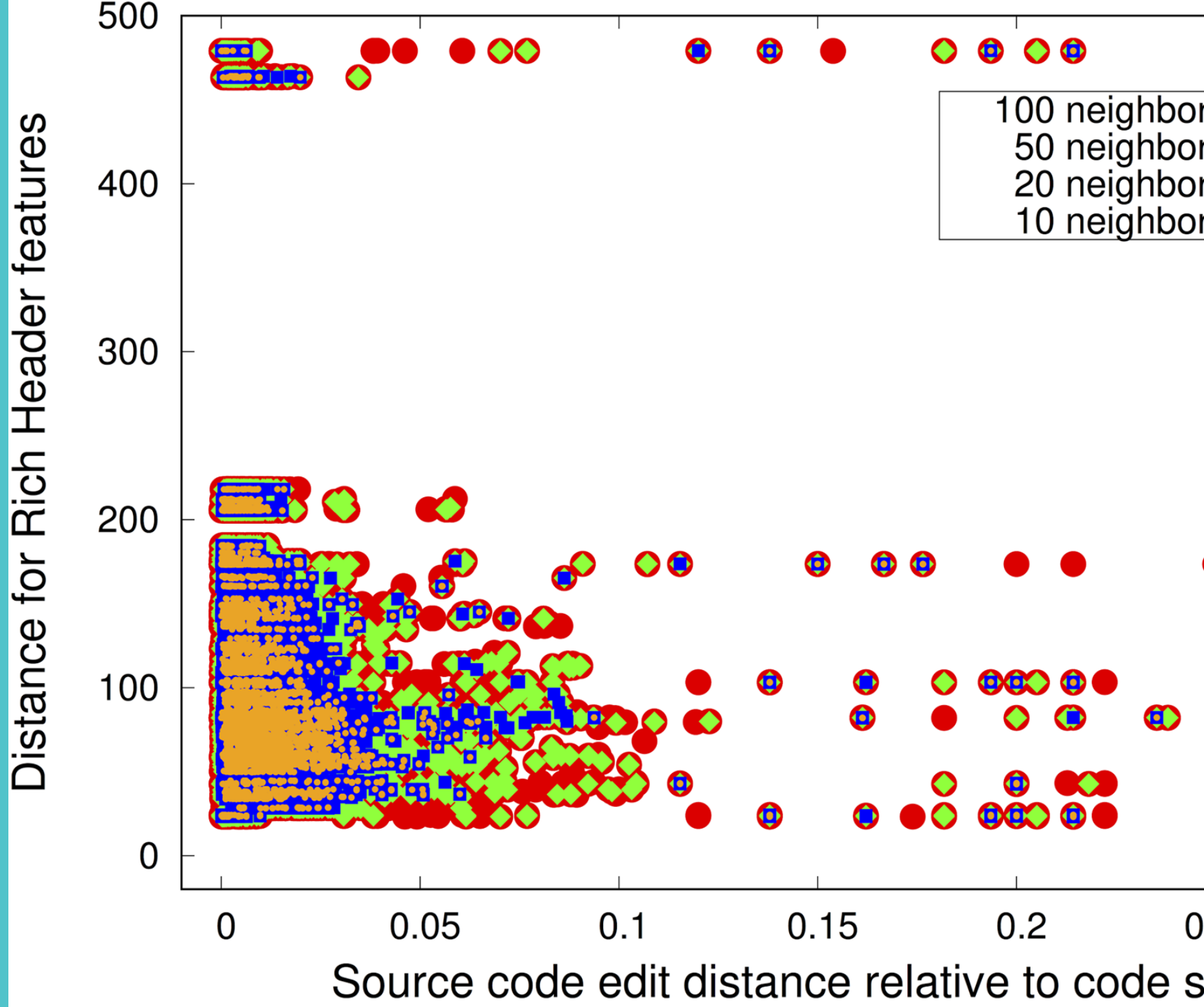
# Validation

Distribution of positions of APT1 samples when querying APT1



# Validation

Correlation of IDA generated code across 1 million random samples. Using entropy of source code.



# 07: Conclusion

Where do we go from here?

# Future Work / Remaining Questions

Rich Header is valuable for triage but future work remains:

- **ProdID:** What are the true mappings?
- **Checksum:** Why is the checksum designed as it is?
- **Purpose:** What was the origin intention, why it is maintained, why is it so hidden?
- **Combine:** Individual triage methods can be overcome. This technique should be combined with other algorithms

# Take-Aways

## Rich Header is valuable for triage

- **Quick Detection:** Identify packed and post modified binaries
- **Similarity Matching:** Find binaries with same functionality
- **Build Environment Fingerprinting:** Develop an understanding of the machine that built the binary
  - Data point for attribution
  - Starting point to identify other malware the actor built

**Thank you**

- George Webster
- Christian von Pentz
- Marcel Schumacher
- Julian Kirsch
- Apostolis Zarras
- Claudia Eckert

**Zachary Hanif**
**Capital One**

**Bojan Kolosnjaji**
**Technical University of Munich**

# Releasing The Rich Header Extractor

Apache2 License
Docker Service
Ready to use with Holmes

holmesprocessing.github.io
@holmesprocess

```
    DANS = 0x536E6144 # 'DanS' as dword
    RICH = 0x68636952 # 'Rich' as dword

    try:
        rich_data = pe.get_data(0x80, 0x80)
        current_pos = 0x80+0x80
        if len(rich_data) != 0x80:
            return None
        data = list(struct.unpack("<32I", rich_data))
    except:
        return None

    # the checksum should be present 3 times after the DanS signature
    checksum = data[1]
    if (data[0] ^ checksum != DANS
        or data[2] != checksum
        or data[3] != checksum):
        return None
    d['checksum'] = checksum

    # add header values
    headervalues = []
    headerparsed = []
    data = data[4:]
    found_end = False
    while not found_end:
        for i in xrange(len(data) // 2):

            # Stop until the Rich footer signature is found
            if data[2 * i] == RICH:
                found_end = True
                # it should be followed by the checksum
                if data[2 * i + 1] != checksum:
                    print('Rich Header corrupted')
                break

            # header values come by pairs
            temp1 = data[2 * i] ^ checksum
            temp2 = data[2 * i + 1] ^ checksum
            headervalues.extend([temp1, temp2])
            headerparsed.append({'id': temp1 >> 16,
                                 'version': temp1 & 0xFFFF,
                                 'times_used': temp2})
```