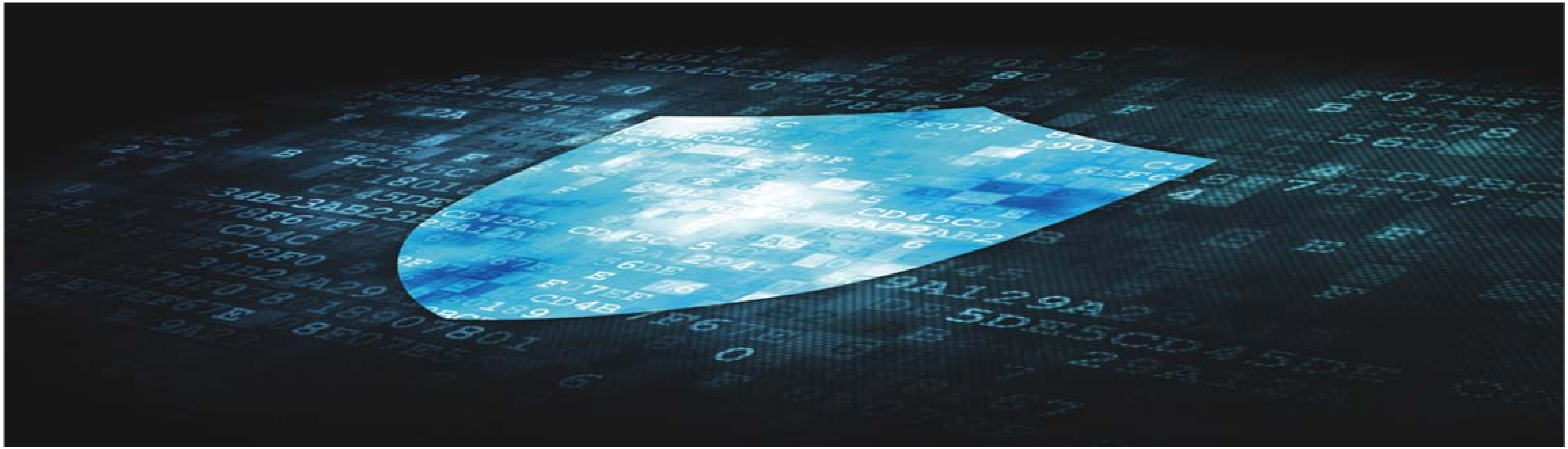
2019-12-06 | Botconf, Bordeaux

YARA-Signator

Automated Generation of Code-based YARA Rules

Felix Bilstein
@fxb_b

Daniel Plohmann
@push_pnx



Fraunhofer FKIE

Introduction

- Felix Bilstein
  - Student @ University Bonn, Research Assistant @ Fraunhofer FKIE
- Daniel Plohmann
  - Reverse Engineering & Malware Research @ Fraunhofer FKIE

Fraunhofer FKIE

The

Agenda

Agenda

- Motivation
- Approach
- Evaluation
- Future Work

Our

Motivation

Motivation
YARA

- Classification of malware is essential for effective processing, e.g.
  - Analysis speedup through contextualization
  - Automated extraction of configurations
- YARA is a de-facto standard: a highly efficient pattern matching engine and rule language
  - „Easy to learn, hard to master“
  - Effective rule creation is still „expensive“ (experience + expert knowledge)

Motivation
Rules: State of the Art

- Currently, most publicly available rules are majorily composed by (text) strings:
  - Rule sets: Mike Worth [1], Florian Roth [2], YaraRules [3], deadbits [4], [redacted], …
    - Files: 2,516, Rules: 26,515
    - 73,295 (75.25%) text strings, 23,367 (23,99%) bytes, 736 (0,76%) regex
- Potential advantages of code-based (byte) rules:
  - Typically robust when targeting the „right“ code areas
  - Harder to circumvent by malware authors(?)
  - Automation scales better than manual effort :)

[1] https://github.com/mikesxrs/Open-Source-YARA-rules
[2] https://github.com/Neo23x0/signature-base
[3] https://github.com/Yara-Rules/rules
[4] https://github.com/deadbits/yara-rules

Motivation
Automated Rule Generation / Related Work

- Limited number of tools for automated rule generation:
  - Blichmann: vxsig [1] / Zaddach&Graziano: BASS [2]
  - Roth: yarGen [3]
  - Doman: YaBin [4]

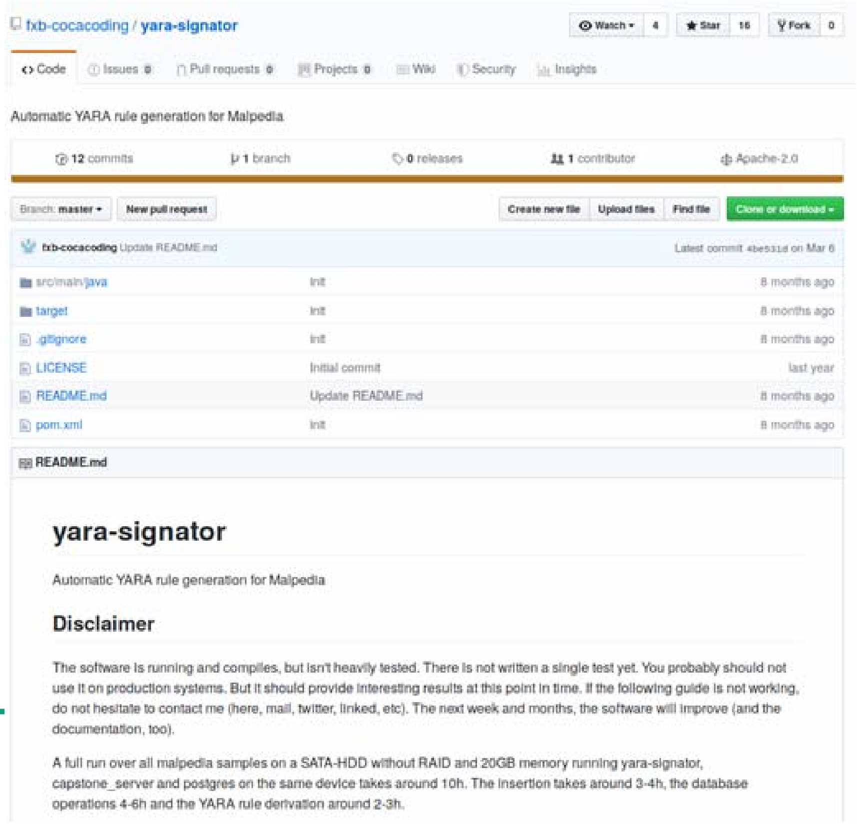
[1] https://github.com/google/vxsig
[2] https://github.com/Cisco-Talos/BASS
[3] https://github.com/Neo23x0/yarGen
[4] https://github.com/AlienVault-OTX/yabin

Fraunhofer
FKIE

Motivation

YARA-Signator

- Practical usage example of the data contained in Malpedia [1]:
  - Started as BA thesis [2,3], continued as MA lab
  - Automated creation of YARA rules!



[1] https://malpedia.caad.fkie.fraunhofer.de
[2] http://cocacoding.com/papers/Automatic_Generation_of_code_based_YARA_Signatures.pdf
[3] https://github.com/fxb-cocacoding/yara-signator

Approach

Approach
Objectives

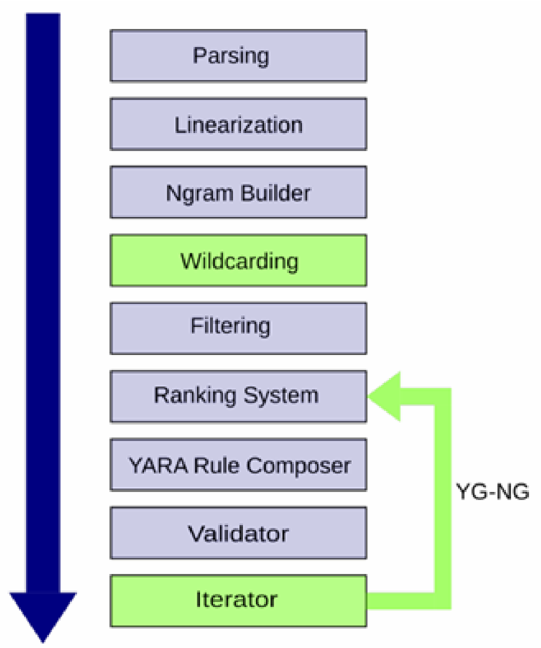
- Goal:
  - Generate accurate YARA rules for as many families in Malpedia as possible
  - Automate YARA string (i.e. byte sequence) selection procedure
  - Quality assurance
- Method:
  - Disassembly -> Shingling -> Aggregation
- Result: YARA-Signator [1]

[1] https://github.com/fxb-cocacoding/yara-signator

Fraunhofer
FKIE

Approach
Modular Procedure

- Approach:
  - Disassemble all unpacked/dumped samples in Malpedia using SMDA [1], then…

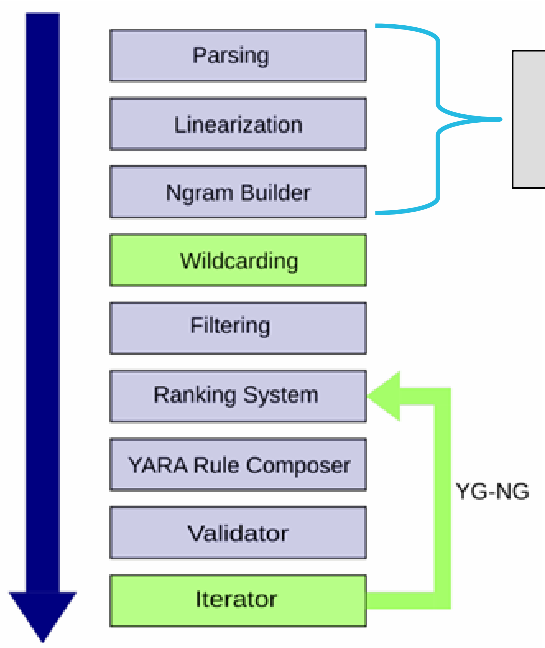


[1] https://github.com/danielplohmann/smda

Fraunhofer FKIE

Approach
Modular Procedure

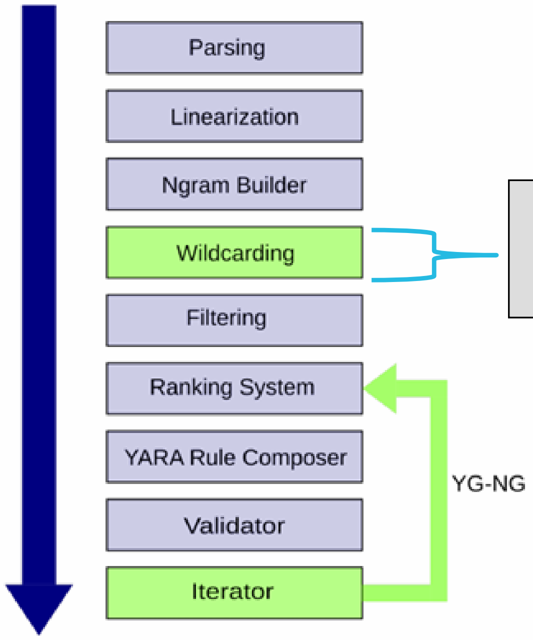
- Approach:
  - Disassemble all unpacked/dumped samples in Malpedia using SMDA, then…



SMDA produces disassembled ICFG in JSON -
but YARA matches linear byte sequences.
-> Linearization & Tokenization into 4/5/6/7 instruction Ngrams

Approach
Modular Procedure

- Approach:
  - Disassemble all unpacked/dumped samples in Malpedia using SMDA, then…



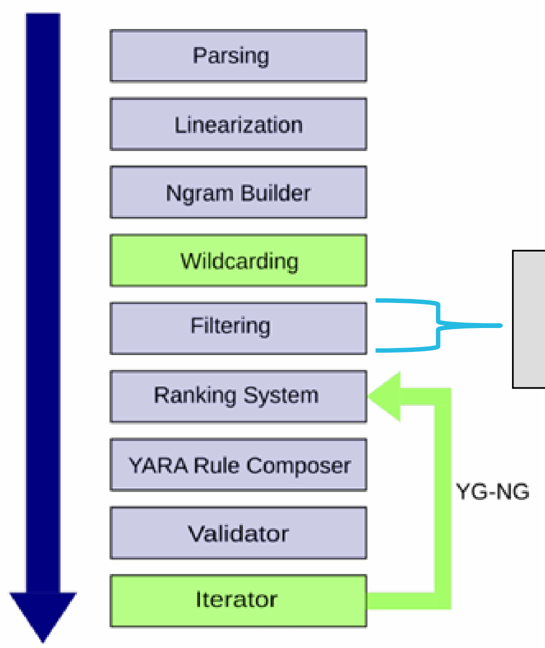
Unpacked / Dumped samples are typically affected by relocations.
Wildcard those addresses for better rule generalization.
Target CFG instructions and data pointer references.

```
$sequence_wildcarding_disabled = { 8b0da4724100 8b5dfc 50 e8c1feffff }
// 8b0da4724100            | mov ecx, dword ptr [0x4172a4]
// 8b5dfc                  | mov ebx, dword ptr [ebp - 4]
// 50                      | push eax
// e8c1feffff              | call 0xfffffec6

$sequence_wildcarding_cfg_ptr = { 8b0d???????? 8b5dfc 50 e8???????? }
// 8b0d????????            | mov ecx, dword ptr [????????]
// 8b5dfc                  | mov ebx, dword ptr [ebp - 4]
// 50                      | push eax
// e8????????              | call ????????
```

Approach
Modular Procedure

- Approach:
  - Disassemble all unpacked/dumped samples in Malpedia using SMDA, then…

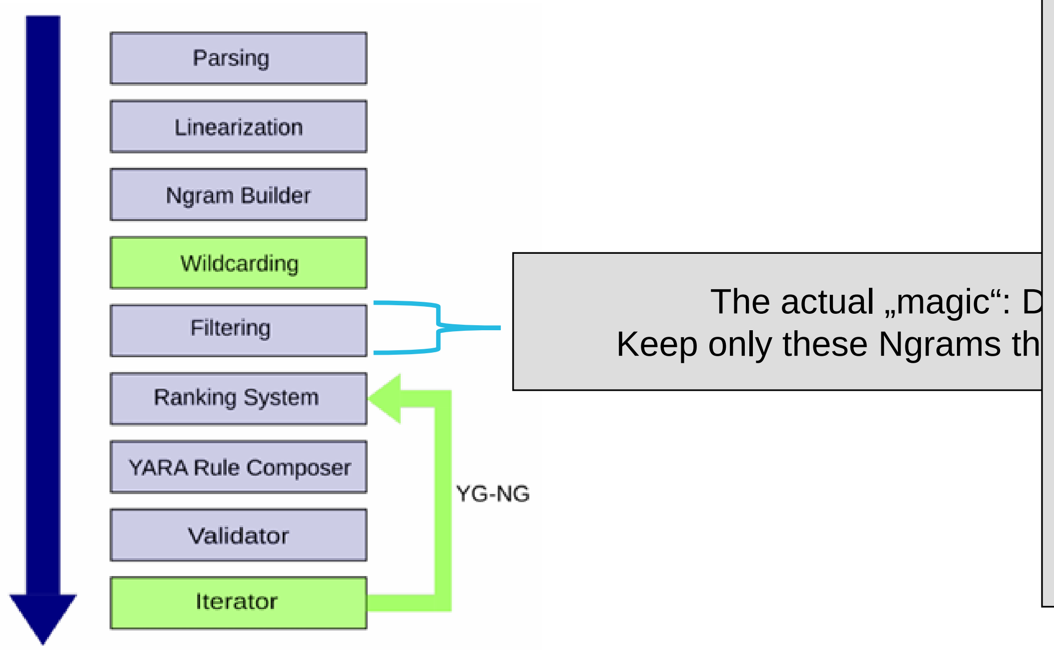


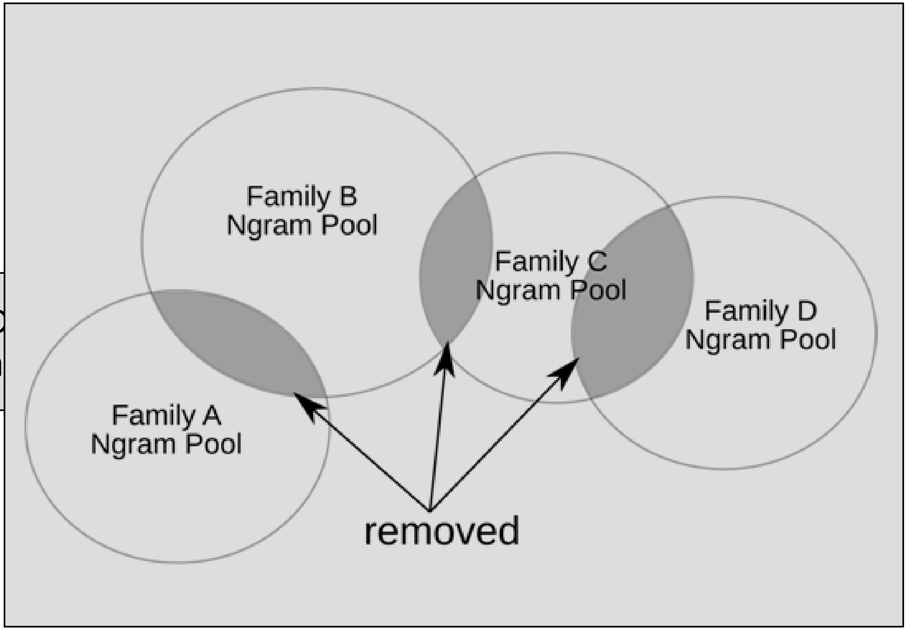
The actual „magic“: Data deduplication ☺
Keep only these Ngrams that occur in a single family.

Fraunhofer FKIE

Approach
Modular Procedure

- Approach:
  - Disassemble all unpacked/dumped samples in Malpedia using SMDA, then…



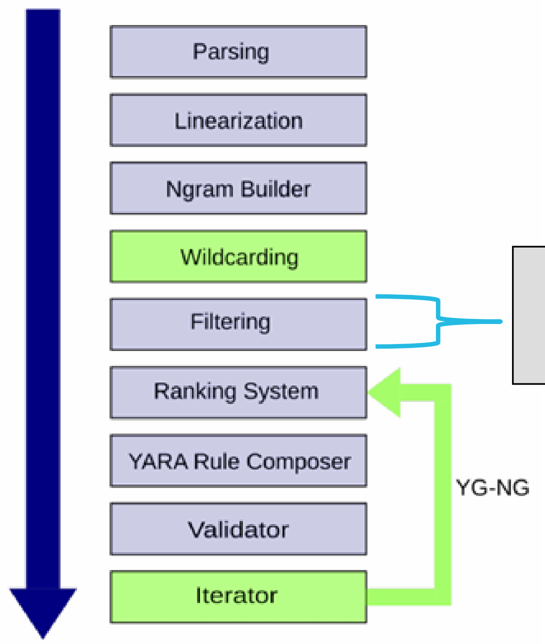


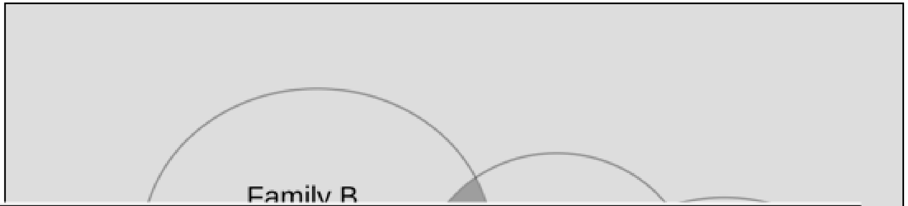
Fraunhofer FKIE

Approach

Modular Procedure

- Approach:
  - Disassemble all unpacked/dumped samples in Malpedia using SMDA, then…





ngram-size	min	25%	50%	75%	max	mean
n = 4	0.035	0.565	0.752	0.893	1.0	0.720
n = 5	0.039	0.606	0.787	0.926	1.0	0.754
n = 6	0.041	0.641	0.823	0.945	1.0	0.778
n = 7	0.041	0.669	0.838	0.957	1.0	0.797
n = 4-7	0.039	0.621	0.796	0.928	1.0	0.760

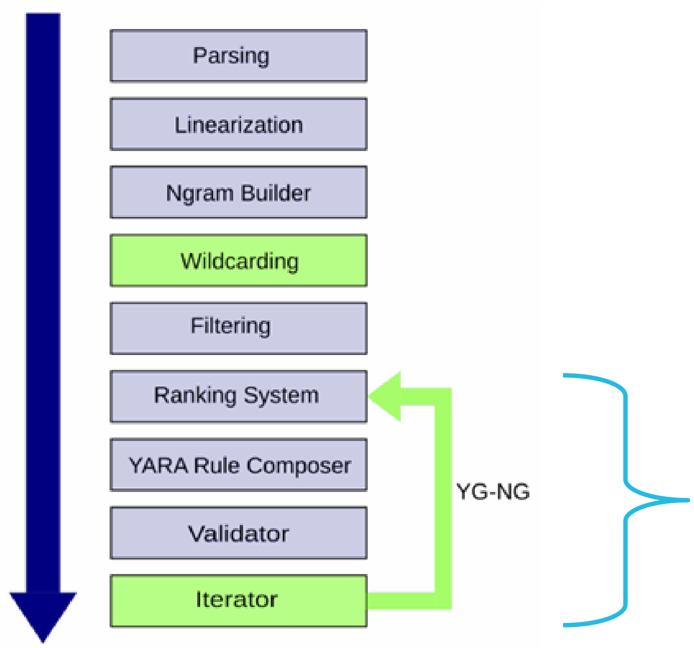
Table: Percentage of unique Ngrams, aggregated on family level [1]

[1] http://cocacoding.com/papers/Automatic_Generation_of_code_based_YARA_Signatures.pdf

Fraunhofer FKIE

Approach
Modular Procedure

- Approach:
  - Disassemble all unpacked/dumped samples in Malpedia using SMDA, then…



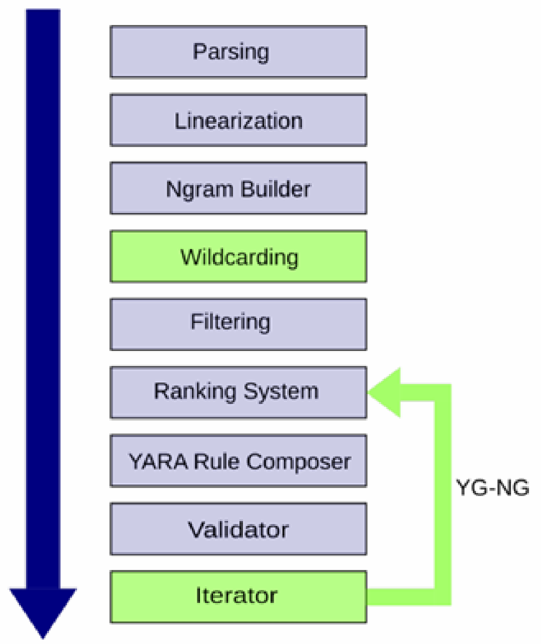
Among candidate Ngrams per family, select those with best coverage.
Require 7 sequences per sample.
Compose and validate YARA rules.
First round: Remove FP sequences („any of them“).
Second round+: Increase coverage by selecting more sequences.

Fraunhofer FKIE

Approach

Modular Procedure

- Approach:
  - Disassemble all unpacked/dumped samples in Malpedia using SMDA, then…




```
rule win_corebot_auto {
  meta:
    author = "Felix Bilstein - yara-signator at cocacoding dot com"
    description = "autogenerated rule brought to you by yara-signator"
    tool = "yara-signator 0.2a"
    malpedia_version = "20190620"
    malpedia_license = "CC BY-SA 4.0"
    malpedia_sharing = "TLP:WHITE"
  strings:
    $sequence_0 = { 7410 85c0 740c 50 }
    $sequence_1 = { 895e0c 8b03 894604 83e880 }
      [...]
    $sequence_9 = { 8b00 894614 83e880 894618 }
  condition:
    7 of them
}
```

Approach
Implementation & Performance

- Implementation as modular framework using:
  - Java
  - Postgres
  - YARA
- Performance (full run on data set explained in the following):
  - Hardware: Intel I7, 32GB RAM, 1 HDD+SSD

	Unmasked (in hours)	Wildcarded (in hours)
Parsing, Linearization, Ngrams, Wildcarding	6.5	5.5
Filtering	2.5	2
First Round	4	2
Following Rounds	1	1
Total	14	10.5

Evaluation

Evaluation
Data Sets

- Malpedia [1]
  - Curated, free, high-quality malware corpus for research
  - Snapshot: 2019-10-21 09:13:52 (commit: d006d14)
- empty_msvc [2]
  - Empty Visual Studio Projects for all versions (VS6-VS2019), built with different bitness and compiler settings
  - Ground-truth for the most common statically linked code

[1] https://malpedia.caad.fkie.fraunhofer.de
[2] https://github.com/danielplohmann/empty_msvc

Fraunhofer FKIE

Evaluation
Code Statistics

- Malpedia [1] (commit: d006d14, date: 2019-10-21)

	Families	Samples
Total	1,447	4,237 (8,508 Files)
Processable (unpacked)	1,085	3,159 (4,575 Files)
Detectable	949	2,916 **(3,978 Files)**

- Code Statistics:

	Averaged over Families					
	Min	25%	50%	75%	Max	Total
Functions	1	136	394	855	19,126	3,092,621
Instructions	2	7061	20,775	51,340	1,311,391	157,806,663

[1] https://malpedia.caad.fkie.fraunhofer.de

Evaluation

Ngram Statistics

- Building Ngrams of length 4-7 instructions
- Data Reduction through Ngram aggregation:
  - Already uniquified per sample while parsing

	Raw	Distinct (over all samples)	Aggregated (occurrence in one family only)
Unmasked	519,242,107	305,473,086	290,209,974
Wildcarded	476,110,027	191,035,382	170,868,100

- Observations:
  - Unique wildcarded Ngrams are significantly less compared to unmasked
  - Family-based code-isolation leaves way larger pool of Ngrams than initially expected

Fraunhofer FKIE

Evaluation

Rule Statistics

- YARA Signator Output: 949 Rules

Total sequences in all rules: 11,825

Wildcarded: 5,765 (48,75%)

736/949 rules (77.56%)

min	2
25%	10
50%	10
75%	10
max	235

Sequences per rule

```
rule yara-signator {
  meta:
    description = "rule statistics"
strings:
    $sequence_0 = { 7410 85c0 740c 50 }
    $sequence_1 = { 895e0c 8b03 894604 e8???????? 83e880 }
      [...]
    $sequence_9 = { 8b00 894614 83e880 894618 }
  condition:
    7 of them
}
```

	Min	25%	50%	75%	Max
Bytes	4	14	18	23	70

Bytes per sequence

Fraunhofer FKIE

Evaluation
Classification Performance

	Families	Samples
Total	1,447	4,237 (8,508 Files)
Processable	1,085	3,159 (4,575 Files)
Detectable	949	2,916 (3,978 Files)

- Hits:

	True	False
Positive	3,759	48
Negative	4,482	219

- Stats:
  - PPV / Precision: 0.987
  - TPR / Recall: 0.945
  - **F1: 0.966**

- Rule Performance:
  - Rules without FPs: 924
  - Rules without FNs: 844
  - „Clean“ Rules: 840

- Reasons for…
  - False Positives:
    - Disassembly inaccuracies
    - Groundtruth / Labeling
  - False Negatives:
    - Modules excluded from procedure

Evaluation

False Positive Analysis vs. Avast 10TB Goodware Data Set

- Avast generously supported our research by running rules against one of their clean data sets.
  - Previous performance evaluation (snapshot July 2nd 2019):
    - Rules for 877/1320 families
    - F-Score: 0.977

- False Positive Analysis vs. Avast Goodware Data Set (10TB):
  - Total FPs: 129,267
  - From 100/877 YARA rules trigger false positives.
    - 23 of 100 are below 10 FPs
    - 67 of 100 are below 100 FPs
    - 87 of 100 are below 1,000 FPs
    - 98 of 100 are below 10,000 FPs
  - YARA signature for "~~win quantloader~~" triggers 51,819 (FP) hits on the data set
  - Rules for which significant FPs are reported get removed from Malpedia

Fraunhofer
FKIE

Evaluation

Interesting FP: win.tinynuke -> win.unidentified_068

```
$ yara -C malpedia_auto.yac malpedia/win.unidentified_068 -r -s | sort

win_tinynuke_auto /malpedia/win.unidentified_068/[redacted]_dump7_0x00400000
                      0x7431:$sequence_2: 89 44 24 1C 2B 58 34 83 3F 00 74 5F 8D 47 04 89 44 24 14
                      0x7446:$sequence_3: 83 F8 08 72 46 83 C0 F8 D1 E8 89 44 24 10 BA 00 00 00 00 74 36
                      0x746b:$sequence_5: 83 F8 03 74 13 83 F8 0A 75 15 8B 07 03 06
                      3 4B 0C
                      0 8B 06 03 C1 50
                      0 8D 44 24 14
```

```
seg000:00407431 89 44 24 1C                 mov     [esp+38h+var_1C], eax
seg000:00407435 2B 58 34                    sub     ebx, [eax+34h]
seg000:00407438 83 3F 00                    cmp     dword ptr [edi], 0
seg000:0040743B 74 5F                       jz      short loc_41749C
seg000:0040743D
seg000:0040743D              loc_41743D:                             ; CODE XREF: sub_417418+82↓j
seg000:0040743D 8D 47 04                    lea     eax, [edi+4]
seg000:00407440 89 44 24 14                 mov     [esp+38h+var_24], eax
seg000:00407444 8B 00                       mov     eax, [eax]
seg000:00407446 83 F8 08                    cmp     eax, 8
seg000:00407449 72 46                       jb      short loc_417491
seg000:0040744B 83 C0 F8                    add     eax, 0FFFFFFF8h
seg000:0040744E D1 E8                       shr     eax, 1
seg000:00407450 89 44 24 10                 mov     [esp+38h+var_28], eax
seg000:00407454 BA 00 00 00 00              mov     edx, 0
seg000:00407459 74 36                       jz      short loc_417491
seg000:0040745B
seg000:0040745B              loc_41745B:                             ; CODE XREF: sub_417418+77↓j
seg000:0040745B 0F B7 44 57 08              movzx   eax, word ptr [edi+edx*2+8]
seg000:00407460 8B C8                       mov     ecx, eax
seg000:00407462 C1 E8 0C                    shr     eax, 0Ch
seg000:00407465 81 E1 FF 0F 00 00           and     ecx, 0FFFh
seg000:0040746B 83 F8 03                    cmp     eax, 3
seg000:0040746E 74 13                       jz      short loc_417483
seg000:00407470 83 F8 0A                    cmp     eax, 0Ah
seg000:00407473 75 15                       jnz     short loc_41748A
seg000:00407475 8B 07                       mov     eax, [edi]
seg000:00407477 03 06                       add     eax, [esi]
```


Fraunhofer FKIE

Evaluation

Interesting FP: win.tinynuke -> win.unidentified_068

```
v1 = *a1;
v2 = *(a1 + 4);
v20 = *a1 + *(*a1 + 60);
for ( i = v1 - *(v20 + 52); *v2; v2 += *v18 )
{
  v18 = v2 + 4;
  v4 = *(v2 + 4);
  if ( v4 >= 8 )
  {
    v16 = (v4 - 8) >> 1;
    v5 = 0;
    if ( v16 )
    {
      do
      {
        v6 = *(v2 + 2 * v5 + 8) >> 12;
        v7 = *(v2 + 2 * v5 + 8) & 0xFFF;
        if ( v6 == 3 )
        {
          *(*a1 + *v2 + v7) += i;
        }
        else if ( v6 == 10 )
        {
          *(v7 + *a1 + *v2) += i;
        }
        ++v5;
      }
      while ( v5 < v16 );
    }
  }
}
v8 = *(a1 + 8);
v17 = v8;
seg000:00407477 03 06
```

```
DWORD WINAPI Payload(InjectDat
{
    IMAGE_DOS_HEADER *dosHeader
    IMAGE_NT_HEADERS *ntHeaders
    IMAGE_BASE_RELOCATION *base

    size_t delta = (size_t) inj

    while(baseRelocation->Virtu
    {
        if(baseRelocation->SizeOfBlock >= sizeof(IMAGE_BASE_RELOCATION))
        {
            DWORD count = (baseRelocation->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) / sizeof(WORD);
            WORD *pWord = (WORD *) (baseRelocation + 1);

            for(DWORD i = 0; i < count; ++i)
            {
                DWORD type = pWord[i] >> 12;
                DWORD offset = pWord[i] & 0xfff;

                switch(type)
                {
                    case IMAGE_REL_BASED_HIGHLOW:
                    {
                        DWORD *patchAddress;

                        patchAddress = (DWORD *) (((DWORD) injectData->base) + baseRelocation->VirtualAddress + offset);
                        *patchAddress += (DWORD) delta;
                        break;
                    }
```

Found reuse of previously „unique“ code, yay!

Meanwhile identified by Proofpoint as „Buer“ [1] (loader)

[1] https://www.proofpoint.com/us/threat-insight/post/buer-new-loader-emerges-underground-marketplace

Fraunhofer FKIE

Evaluation
Discussion / Lessons Learned

- Lots of family-unique Ngrams available!
  - This massively benefits rule generation *(probably also code similarity analysis)*
- Input data quality is essential:
  - Disassembly errors -> False Positives
  - Insufficient example coverage leads to inferior rules:
    - 64bit
    - Static linking: Delphi, Go
- Biggest rule quality improvement:
  - In Ngram selection process, exclude overlaps!

[1] https://malpedia.caad.fkie.fraunhofer.de
[2] https://github.com/danielplohmann/empty_msvc

Fraunhofer FKIE

Future Work

Fraunhofer
FKIE

Future Work

Potential Improvements

- Support more architectures (e.g. ARM, MIPS) or input formats
- „Daemonization"
  - Periodic (daily/weekly?) runs for Malpedia
  - Work on coverage maximization for prevalent families
- Further evaluation
  - Minimize signatures (less sequences, …)
  - Compare usage of raw bytes versus instruction ngrams?

Fraunhofer FKIE

Thank You for Your Attention!

Felix Bilstein
fxb@cocacoding.com
 @fxb_b

Daniel Plohmann
daniel.plohmann@fkie.fraunhofer.de
@malpedia @push_pnx

Fraunhofer
FKIE

Evaluation

BONUS: Instruction Statistics

- Most common mnemonics
  - Difference: 32bit has extensive stack usage, not so 64bit (among other things because of `fastcall` calling convention)
  - Apart from order, mostly the same for 32bit & 64bit

		32bit		64bit	
		Count	%	Count	%
1	mov	40599533	28.408826	5046767	40.256927
2	push	22027485	15.413355	214419	1.710372
3	call	12034871	8.421194	1105853	8.821141
4	pop	7130588	4.989507	211099	1.683889
5	cmp	6561628	4.591387	624762	4.983586
6	lea	6099424	4.267967	952356	7.596730
7	add	5354167	3.746486	448923	3.580958
8	je	5208274	3.644400	456229	3.639236
9	test	4632029	3.241183	466811	3.723647
10	jmp	4165446	2.914699	417208	3.327975
11	xor	4027780	2.818370	498826	3.979023
12	jne	3667381	2.566187	347884	2.774993
13	dec	3504595	2.452280	31986	0.255145
14	ret	2862709	2.003131	196066	1.563974
15	inc	2072344	1.450087	82619	0.659033

Most common mnemonics

Fraunhofer FKIE

Evaluation

BONUS: Instruction Statistics

- Semantic Class Distribution
  - 32bit: extensive stack usage
  - 64bit: fast-call -> memory ops
- Classes:

  M: Memory
  C: CFG
  S: Stack
  A: Logic/Arithmetic
  X: Extended (MMX, SSE, …)
  F: Float
  N: Nop
  P: Privileged
  Y: Crypto
  V: VMX

	32bit		64bit	
	Count	%	Count	%
M	48,708,977	33.569453	6,296,326	49.547899
C	42,619,831	29.372910	3,952,593	31.104279
S	29,570,122	20.379258	425,962	3.352038
A	22,443,025	15.467376	1,838,024	14.464027
X	600,975	0.414182	109,975	0.865430
F	432,431	0.298025	155	0.001220
N	420,548	0.289835	42,351	0.333274
P	290,675	0.200329	37,342	0.293857
Y	12,515	0.008625	4826	0.037977
V	10	0.000007	-	-

Semantic Classes

Fraunhofer FKIE

Evaluation

BONUS: Instruction Statistics

- Instruction Length Distribution
  - Less 1 Byte instructions on 64bit:
    - Because 0x4? Instructions used as 64bit „marker“
- Operand Count Distribution
  - 3+4 operands mostly found in extended instruction sets (MMX, SSE, …)

# Operands	Count	%
0	3,869,620	2.452127
1	67,833,700	42.985321
2	85,794,407	54.366784
3	305,727	0.193735
4	3209	0.002034

Operand Count Statistics

	32bit		64bit	
	Count	%	Count	%
1	29,816,283	20.548908	473,392	3.725280
2	40,671,243	28.029974	2,256,002	17.753236
3	32,342,763	22.290118	2,690,095	21.169259
4	9,471,707	6.527750	1,994,185	15.692910
5	17,956,035	12.375014	2,938,995	23.127936
6	10,083,506	6.949392	758,372	5.967883
7	3,635,857	2.505775	803,522	6.323184
8	647,009	0.445908	602,222	4.739087
9	25,148	0.017332	120,774	0.950411
10	396,449	0.273226	29,027	0.228423
11	48,822	0.033647	15,484	0.121849
12	1,134	0.000782	25,149	0.197906
13	1,217	0.000839	174	0.001369
14	968	0.000667	101	0.000795
15	968	0.000667	60	0.000472

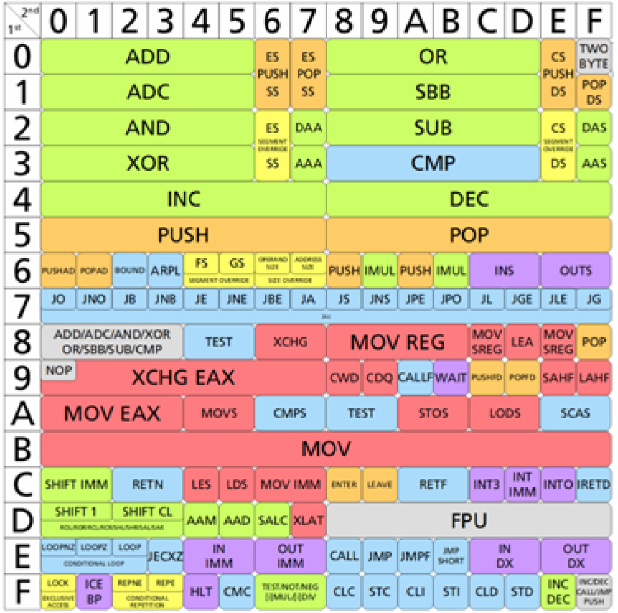
Instruction Length Statistics (in Bytes)

Fraunhofer FKIE

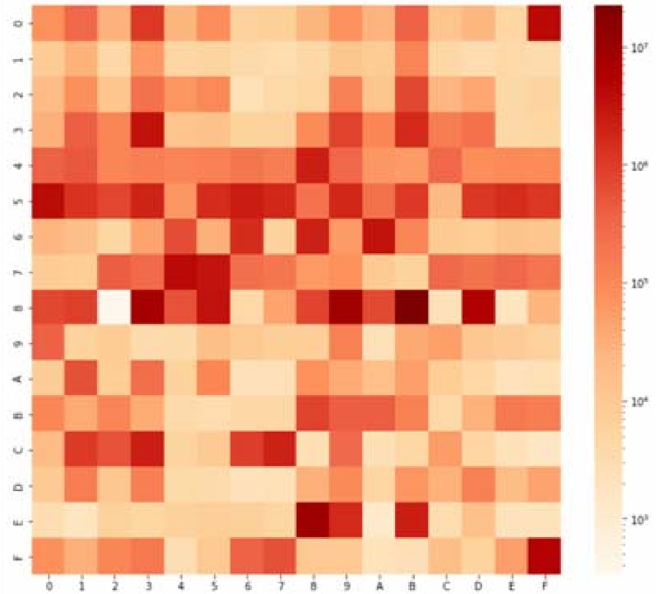
Evaluation

BONUS: Instruction Statistics

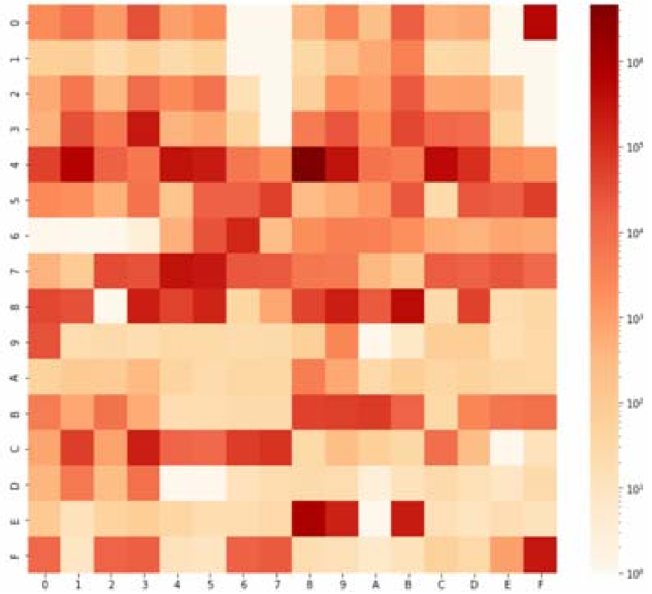
- Instruction First Byte Heatmaps



Reference (32bit) [1]



Heatmap (32bit)



Heatmap (64bit)

[1] https://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/x86_opcode_structure_and_instruction_overview.pdf

Fraunhofer FKIE