Darmstadt University of Applied Sciences, da/sec Security Group

Towards Exact and Inexact Approximate Matching of Executable Binaries

DFRWS-EU 2019, Oslo, Norway

Lorenz Liebler, Harald Baier
Fuzzy Hashing

General

- a.k.a Approximate Matching: is a *similarity preserving hash function*
- in contrary to cryptographic hash functions → determines similarity of two files
- introduced more than a decade ago → deal with spam → forensic challenges
- simple to implement, few computational resources
Overview and History


Fuzzy Hashing

Overview and History

→ especially in the field of malware or binary analysis
Fuzzy Hashing

Schemes

Internal implementations differ heavily

- Context-Triggered Piecewise Hashing (ssdeep, mrsh-v2)
- Statistically Improbable Features (sdhash)
- N-Grams (tlsh)

Simplified overview similar to Ren, Liwei [21] (DFRWS EU 2015):

<table>
<thead>
<tr>
<th>1st Model</th>
<th>2nd Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>File / Binary</td>
<td>Feature Selection</td>
</tr>
<tr>
<td></td>
<td>Features</td>
</tr>
<tr>
<td></td>
<td>Digest Generation</td>
</tr>
<tr>
<td></td>
<td>Digest</td>
</tr>
<tr>
<td></td>
<td>Comparison</td>
</tr>
</tbody>
</table>

- **ssdeep**: chunks of sequences (splitted string)
- **mrsh-v2**: chunks of sequences (extracted by PRF)
- **sdhash**: bag of 64-byte blocks (selected by entropy)
- **tlsh**: bag of triplets (selected from all 5-grams)

- **ssdeep**: mapped chunks into 80 byte digest
- **mrsh-v2**: chunks hashed into Bloom filter
- **sdhash**: blocks mapped into Bloom filter
- **tlsh**: mappend into 32 byte container

- **ssdeep**: Levenshtein distance (0-100)
- **mrsh-v2**: Hamming distance (0-100)
- **sdhash**: Hamming distance (0-100)
- **tlsh**: Distance score (0-1000+)
Fuzzy Hashing

Binary Analysis in Academia (Pagani et al. [19])

Fabio Pagani, Matteo Dell’Amico, and Davide Balzarotti. Beyond precision and recall: Understanding uses (and misuses) of similarity hashes in binary analysis.

- no academic consensus about usefulness
- different evaluation datasets lead to different conclusions for same approaches (e.g., ssdeep)
- avoid: yet another large scale experiment
- inspect reasons for results (*why*; not *if*):
  → four different schemes
  → in *three binary analysis case studies*
Fuzzy Hashing

Scenarios - Pagani et al. [19]
Fuzzy Hashing

Scenarios - Pagani et al. [19]

- **Library Identification**: detecting embedded object files inside a binary
  - 1.1 Object-to-Program Comparison:
    → whole executable (.o)
    → .text segment only.
  - 1.2 Impact of Relocation considers
    → relocations performed by linker / dynamic loader
    → original and relocated object file / final executable
II Re-Compilation: detection of the same program after Re-Compilation

II.1 Effect of Compiler Flags
(same compiler; i.e., O0, O1, O2, O3, Os)

II.2 Effect of Different Compilers
Fuzzy Hashing

Scenarios - Pagani et al. [19]

III Program Similarity: three tests which consider adaptations to the underlying code

III.1 Small Assembly Differences:
→ randomly inserts an increasing amount of NOPs
→ increasing amount of instructions are swapped

III.2 Minor Source Code Modifications:
→ Different Comparison Operator
→ New Condition
→ Change a constant value

III.3 Malware Code Modifications (Mirai, Grum):
→ C2 Domain Adaptation
→ Evasion and New Functionality
the distinction between data and code is of crucial importance

even small changes on the (source) code / additional insertions → influence the overall binary and code structure in a broad way
→ especially has a great impact on CTPH-based approaches
→ similarity is not just a consequence of the size of the change

Summarized, sdhash and tlsh clearly outperformed CTPH-based schemes.
→ Each of both have their strengths and weaknesses in different disciplines.

CTPH (ssdeep) - the de-facto industry standard — is not very well suited to binary analysis in general
Approach

RQs

**mrsh-mem**
→ approxis x86/x64 instruction carver
→ interfaced with mrsh-v2
→ bulk extraction / identification of code

▶ What impact has the discrimination of code and data?
▶ Could we utilize an additional layer of approximate disassemble?
▶ What is the actual improvement in the case of a CTPH-based approaches?
Could we improve the performance only by refining the feature selection?

---

**mrsh-mem** [16]: Lorenz Liebler and Frank Breitinger. *mrsh-mem: Approximate matching on raw memory dumps.* In International Conference on IT Security Incident Management and IT Forensics, pages 47–64. IEEE, 2018

apx-bin: Utilizing mrsh-mem

**Approach**

Approximate Disassembler: approxis maps bytes to mnemonics

x:= start of current chunk
z:= end of current chunk
m:= start of current sliding window
z:= end of current sliding window

BF Similarity Digest

\[ \text{if } b == 0: \quad \text{fnv-1a}(M[x:m+6]) \]

PRF

\[ b := \text{rolling hash}(M[m:m+6]) \mod b \]

**BF Similarity Digest**

**Approximate Disassembler:**

approxis maps bytes to mnemonics
**Approach**

**apx-bin: Utilizing mrsh-mem**

Original mrsh-mem approach:
- Extraction of **Code**-related fragments only
  → now also **Data**-related
- **No scoring** of the different buffers
  → **scoring** of different streams; use Mnemonic- (M) and Byte-Stream (B)

---

BF Similarity Digest

If b == 0:

fnv-1a(M[x:m+6])

PRF

b := rolling_hash(M[m:m+6]) % b

Approximate Disassembler:
aproxis maps bytes to mnemonics

x := start of current chunk
m := start of current sliding window
z := end of current chunk
z := end of current sliding window
Pre-Evaluation

Naive Approach

Prove impact of data- or code-related features by adapted score-model:

1. Extract chunks of code and data by parametrization via $\tau_{min}$ and $\tau_{max}$; number of all chunks (from both buffers) defined as $z$

2. Multi-layered extraction - processing mapped buffer of mnemonics (M) and its byte representation (B)

$$sim_{pre} = \min \left( \frac{\left( \sum_{i=1}^{y} f_c(b_i) \right) \cdot 1.5 + \sum_{i=1}^{y} f_c(m_i) }{z}, 100 \right),$$

where $f_c(x) = \begin{cases} 1, & \text{if } \tau_{min} \leq x \leq \tau_{max} \\ 0, & \text{else} \end{cases}$. (1)
Pre-Evaluation

Naive Approach

Prove impact of data- or code-related features by adapted score-model:

1. Extract chunks of code and data by parametrization via $\tau_{min}$ and $\tau_{max}$; number of all chunks (from both buffers) defined as $z$

2. Multi-layered extraction - processing mapped buffer of mnemonics (M) and its byte representation (B)

$$sim_{pre} = \min \left( \frac{\sum_{i=1}^{y} f_c(b_i) \cdot 1.5 + \sum_{i=1}^{y} f_c(m_i)}{z}, 100 \right),$$

where $f_c(x) = \begin{cases} 1, & \text{if } \tau_{min} \leq x \leq \tau_{max} \\ 0, & \text{else} \end{cases}$. (1)

$\tau$-values filter each extracted chunk:

<table>
<thead>
<tr>
<th>$\tau_{max}$-$\tau_{min}$</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>100-0</td>
<td>All chunks</td>
</tr>
<tr>
<td>100-80</td>
<td>Code chunks</td>
</tr>
<tr>
<td>20-0</td>
<td>Data chunks</td>
</tr>
</tbody>
</table>
Pre-Evaluation

Naive Approach

Prove impact of data- or code-related features by adapted score-model:

1. Extract chunks of code and data by parametrization via $\tau_{min}$ and $\tau_{max}$; number of all chunks (from both buffers) defined as $z$

2. Multi-layered extraction - processing mapped buffer of mnemonics (M) and its byte representation (B)

$$sim_{pre} = \min \left( \frac{\sum_{i=1}^{y} f_c(b_i) \cdot 1.5 + \sum_{i=1}^{y} f_c(m_i)}{z}, 100 \right),$$

where $f_c(x) = \begin{cases} 
1, & \text{if } \tau_{min} \leq x \leq \tau_{max} \\
0, & \text{else}
\end{cases}$. (1)
Pre-Evaluation

Scenario I. Library Identification

Object-to-Program (Whole Object File)

Object-to-Program (Text Section)
Pre-Evaluation

Scenario II. Recompilation

Optimization flags

Compiler Variation
Conclusion

▶ prove of **ambivalence** of the different **use cases**
  ▶ **matching code**: primarily detecting used libraries
  ▶ **matching data**: different compilers / configurations
▶ small constant **data** fragments **versus** large amount of **code** per binary
▶ a match on the **byte level** should be considered as more meaningful
▶ we stick to the extraction of sequences
Approach

Results and observations from the pre-evaluation have been used:

- \( \tau \) now defines center bounds of selected chunks (ignore vague chunks)

\[
\begin{align*}
    f_d(x) &= \begin{cases} 
    1, & \text{if } 0 \leq x \leq \tau_{\text{min}} \\
    0, & \text{else} 
    \end{cases} \\
    f_c(x) &= \begin{cases} 
    1, & \text{if } 100 \geq x \geq \tau_{\text{max}} \\
    0, & \text{else} 
    \end{cases}
\]

- Score model

\[
sim_{bm} = \frac{\gamma_d + \gamma_c}{2}, \text{where}
\]

\[
\gamma_d = \min \left( \frac{\left( \sum_{i=0}^{y-1} f_d(b_i) \cdot 2 + \sum_{i=0}^{z-1} f_d(m_i) \right) \cdot 1.5}{2 \cdot \sum_{i=0}^{n} f_d(c_i)}, 0.99 \right),
\]

\[
\gamma_c = \min \left( \frac{\sum_{i=0}^{y-1} f_c(b_i) \cdot 2 + \sum_{i=0}^{z-1} f_c(m_i)}{2 \cdot \sum_{i=0}^{n-1} f_c(c_i)}, 0.99 \right).
\]

sequence of extracted chunks represented by their specific code coverage:
\[
\langle c_0, c_1, \ldots, c_{n-1} \rangle
\]

hits are defined by their values of code coverage for matching byte chunks
\[
\langle b_0, b_1, \ldots, b_{y-1} \rangle
\]
and for matching mnemonic chunks
\[
\langle m_0, m_1, \ldots, m_{z-1} \rangle
\]
match either as a sequence of mnemonics and bytes, or as a sequence of mnemonics only
\[
y \leq z
\]
Approach

Results and observations from the pre-evaluation have been used:

- $\tau$ now defines center bounds of selected chunks (ignore vague chunks)

\[
f_d(x) = \begin{cases} 
1, & \text{if } 0 \leq x \leq \tau_{min} \\
0, & \text{else}
\end{cases}
\]

\[
f_c(x) = \begin{cases} 
1, & \text{if } 100 \geq x \geq \tau_{max} \\
0, & \text{else}
\end{cases}
\]

- Score model

\[
sim_{bm} = \frac{\gamma_d + \gamma_c}{2}, \text{where}
\]

\[
\gamma_d = \min \left( \frac{\left( \sum_{i=0}^{y-1} f_d(b_i) \cdot 2 + \sum_{i=0}^{z-1} f_d(m_i) \right) \cdot 1.5}{2 \cdot \sum_{i=0}^{n} f_d(c_i)}, 0.99 \right),
\]

\[
\gamma_c = \min \left( \frac{\sum_{i=0}^{y-1} f_c(b_i) \cdot 2 + \sum_{i=0}^{z-1} f_c(m_i)}{2 \cdot \sum_{i=0}^{n-1} f_c(c_i)}, 0.99 \right).
\]
## Scenario I - 1. Object-to-Program Comparison

<table>
<thead>
<tr>
<th>Alg.</th>
<th>.o</th>
<th>.text</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TPR %</td>
<td>FPR %</td>
</tr>
<tr>
<td>ssdeep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mrsh-v2</td>
<td>11.7</td>
<td>0.5</td>
</tr>
<tr>
<td>sdhash</td>
<td>12.8</td>
<td>0</td>
</tr>
<tr>
<td>tsh</td>
<td>0.4</td>
<td>0.1</td>
</tr>
<tr>
<td>apx-bin</td>
<td>48.9</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Competitive Evaluation

Scenario I - 2. Impact of Relocation

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Average Score (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssdeep</td>
<td>0.0</td>
</tr>
<tr>
<td>tlsh</td>
<td>12.3</td>
</tr>
<tr>
<td>sdhash</td>
<td>26.0</td>
</tr>
<tr>
<td>mrsh-v2</td>
<td>10.67</td>
</tr>
<tr>
<td>apx-bin</td>
<td>22.67</td>
</tr>
</tbody>
</table>

Statically-Linked Binary

Original Object
- ssdeep = 0
- tlsh = 0
- sdhash = 11
- mrsh-v2 = 4
- apx-bin = 33

Relocated Object
- ssdeep = 0
- tlsh = 11
- sdhash = 14
- mrsh-v2 = 4
- apx-bin = 23
Competitive Evaluation

Scenario II Re-Compilation - 1. Optimization Flags

→ no comparisons of same configuration (same flags)
→ zero false positives
→ coloured area: highest score of a false match
Competitive Evaluation

Scenario II Re-Compilation - 2. Different Compilers

→ no comparisons of same configuration (same compiler)
→ zero false positives
→ coloured area: highest score of a false match
Competitive Evaluation

Scenario III Program Similarity - 1. Small Assembly Differences

![Graph showing the comparison of different similarity metrics such as apx-bin, apx-code, apx-data, ssdeep, sdhash, mrsh-v2, and tlsh. The x-axis represents the number of inserted nops or swaps, and the y-axis represents the score. The graph illustrates how the score decreases as the number of inserted nops or swaps increases.]
Competitive Evaluation

Scenario III Program Similarity

III.2 Minor Source Code Modifications

<table>
<thead>
<tr>
<th>Change</th>
<th>ssdeep</th>
<th>mrsh-v2</th>
<th>tlsh</th>
<th>sdhash</th>
<th>apx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>0-100</td>
<td>21-100</td>
<td>99-100</td>
<td>22-100</td>
<td>76-99</td>
</tr>
<tr>
<td>Condition</td>
<td>0-100</td>
<td>22-99</td>
<td>96-99</td>
<td>37-100</td>
<td>83-99</td>
</tr>
<tr>
<td>Constant</td>
<td>0-97</td>
<td>28-99</td>
<td>97-99</td>
<td>35-100</td>
<td>81-99</td>
</tr>
</tbody>
</table>

→ score ranges (min-max)

III.3 Minor Source Code Modifications on Malware

<table>
<thead>
<tr>
<th>Change</th>
<th>ssdeep</th>
<th>mrsh-v2</th>
<th>tlsh</th>
<th>sdhash</th>
<th>apx</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2 domain (r)</td>
<td>M</td>
<td>G</td>
<td>M</td>
<td>G</td>
<td>M</td>
</tr>
<tr>
<td>C2 domain (l)</td>
<td>0</td>
<td>0</td>
<td>97</td>
<td>10</td>
<td>99</td>
</tr>
<tr>
<td>Evasion</td>
<td>0</td>
<td>0</td>
<td>44</td>
<td>13</td>
<td>94</td>
</tr>
<tr>
<td>Functionality</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>0</td>
<td>93</td>
</tr>
</tbody>
</table>

→ real (r) or long (l) domain
→ evasion: anti-debugger, anti-VM techniques
→ functionality: enumerate users on infected system
Conclusion

- reassessed previous research
- demonstrate relevance of feature selection for matching-success
  → apx-bin still relies on CTPH
- outperform several of the existing (also non-CTPH) approaches
- stable scores in all scenarios
Current Work 1/2

- extend approxis-engine:
  - extend carver / approxis-engine
  - interface with other schemes (non-CTPH)

- extend scenarios / break approaches:
  - gather different schemes and evaluation data
  - create online-repository to publish results
  - please feel free to contact me
Current Work 2/2

- benign binaries for different architectures, compilers and languages
- malicious binaries curated by Malpedia [20]
- create a controlled set of obfuscated binaries with the help of Obfuscator-LLVM\(^a\)

\[\begin{align*}
\text{jmpcond} & \text{100pct} & \text{Conversion of conditional jumps} \\
\text{bcf} & \text{5\_10pct} & \text{Bogus Control Flow; across 5 runs} \\
\text{bcf} & \text{1\_100pct} & \text{Bogus Control Flow; across 1 run} \\
\text{noop} & \text{15\_10pct} & \text{max. 15 multi-byte NOPs per Basic Block} \\
\text{noop} & \text{X\_Ypct} & \text{max. X multi-byte NOPs per Basic Block.} \\
\text{fla} & \text{} & \text{Control-Flow-Flattening} \\
\text{split} & \text{5} & \text{Splitting of Basic Blocks; max. 5 splits} \\
\text{sub} & \text{2} & \text{Substitution of commands with 2 runs} \\
\text{dead} & \text{} & \text{Insertion of Dead-Code} \\
\text{split} & \text{dead} & \text{Splitting of Basic-Blocks; max 5 splits; insertion of Dead-Code} \\
\end{align*}\]

\(^a\text{https://github.com/obfuscator-llvm/obfuscator/wiki}\)
Thank you.

lorenz.liebler@h-da.de
https://dasec.h-da.de/staff/lorenz-liebler/
Bibliography I


Bibliography III