FASER: Binary Code Similarity Search through the use of Intermediate Representations

By Josh Collyer, Tim Watson and Iain Phillips
Background
What is Binary Code Similarity Search?

- Essentially an information retrieval task
- Query function and a corpus of other functions, is my query function present within this corpus?
- Has been applied to a wide range of different task:
  - Identifying N-days within software (lots of focus on firmware and IoT)
  - Identifying open source libraries within binaries
  - Identifying function re-use within software and malware
  - Identifying prior reverse engineered functions
This has been around for a while though right?

- Solved with NLP and Graphs (with a growing trend of combos)
- Approaches such as Asm2vec[1], SAFE[2] and GEMINI[3] pushed the field forward in 10's
- Newer NLP approaches have leveraged transformers such as jTrans[4], PalmTree[5] and Trex[6]
  - Either BERT or RoBERTa
  - Pre-trained -> finetuned
- Sometimes cross architecture, usually mono across compilers + compiler options
Our Contribution
Proposed Approach

- Use the advancements in long context transformers by leveraging the LongFormer[7] architecture
  - Different attention mechanism = More input tokens
- Ditch the pre-training and train for the objective directly
  - A more targeted, high performing model
- Use an intermediate representation instead of bytes/disassembly
  - Reduced need to normalisation, small vocab size and inherently cross-architecture
- Use deep metric learning, circle loss, online batch hard mining and dynamic pair generation
  - Leverage the research coming out of facial recognition/image retrieval
Proposed Approach

- Use the advancements in long context transformers by leveraging the LongFormer[7] architecture
  - Different attention mechanism = More input tokens
- Ditch the pre-training and train for the objective directly
  - A more targeted, high performing model
- Use an intermediate representation instead of bytes/disassembly
  - Reduced need to normalisation, small vocab size and inherently cross-architecture
- Use deep metric learning, circle loss, online batch hard mining and dynamic pair generation
  - Leverage the research coming out of facial recognition/image retrieval
The Double Edged Sword - Deep Metric Learning

- Optimisation is driven by a metric - typically distance
- Very frustrating to train
- Usually requires large batch sizes (512+)

Feature Space

Metric Space
Dynamic Pair Mining

- Most prior research using pre-computed pairs/triplets
- General process is:
  - Embed all examples in batch
  - Dynamically make positive and negatives pairs based on labels
  - Generate losses
  - Take the best/worst/mean/something of the losses and use to update network
- Constantly challenge the models weaknesses
Circle Loss[8]

- Used a lot in facial recognition and image retrieval generally
- Uses a circular decision boundary instead of a straight one
- Emphasises suboptimal similarity scores by re-weighting them using a dynamic penalty strength
- Able to deal with large similarity variations at the beginning of training/whilst your learning rate is high
(a) Overview of the Model Architecture Used

(b) Siamese Training Formulation
Dataset Used

- The Dataset-1 and Dataset-Vulnerability from Marcelli et al (2022)[9]
- Dataset-1
  - ClamAV, Curl, nmap, Openssl, Unrar, z3 and zlib compiled using four different version of Clang/GCC over 5 different optimisation levels (1.5M functions once processed)
- Dataset-Vulnerability
  - A 14 OpenSSL CVE’s present within the libcrypto libraries of two firmware images alongside the same library compiled for arm32, mips32, x86 and x86-64.
Experiments
General Function Search

Objective: Given a query function, can the model correctly retrieve the correct function within a search pool of 100 negatives and 1 positive?

- The XM task was used from within Marcelli (2022)
- No constraints on architectures, bitness, compilers or optimisations.
- Hardest and closest to real life

Metrics: Recall@1 (also Precision@1 due to there only being one positive (i.e relevant) function in search pool) & Mean Reripical Rank (MRR)@10 (how far down the ranking the first relevant function is)
## Results

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>R@1</th>
<th>MRR@10</th>
</tr>
</thead>
<tbody>
<tr>
<td>FASER NRM</td>
<td>ESIL Function String</td>
<td>0.51</td>
<td>0.57</td>
</tr>
<tr>
<td>FASER RN</td>
<td>ESIL Function String</td>
<td>0.46</td>
<td>0.53</td>
</tr>
<tr>
<td>GMN [13]</td>
<td>CFG + BoW opc 200</td>
<td>0.45</td>
<td>0.53</td>
</tr>
<tr>
<td>GNN [13]</td>
<td>CFG + BoW opc 200</td>
<td>0.44</td>
<td>0.52</td>
</tr>
<tr>
<td>GNN (s2v) [23]</td>
<td>CFG + BoW opc 200</td>
<td>0.26</td>
<td>0.36</td>
</tr>
</tbody>
</table>
Vulnerability Search

**Objective:** Given a known vulnerable function within the OpenSSL libcrypto library, can the model identify if a given firmware OpenSSL libcrypto library also contains the function?

- Approximately ~1000 functions in the firmware OpenSSL library
- Search-pool is effectively 10 times the size

Results reported are for the NETGEAR R700 (arm32) libcrypto vulnerability search.
## Results

<table>
<thead>
<tr>
<th></th>
<th>NETGEAR R700</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARM</td>
</tr>
</tbody>
</table>
Zero Shot Experiment

<table>
<thead>
<tr>
<th></th>
<th>ARM32</th>
<th>MIPS32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean Rank</td>
<td>Median Rank</td>
</tr>
<tr>
<td>FASER NRM</td>
<td>48;546;964;14</td>
<td>393; 297; 48;30;22;546;170;</td>
</tr>
<tr>
<td></td>
<td>393</td>
<td>297; 251;14;155</td>
</tr>
<tr>
<td>FASER RM</td>
<td>673;292;1004;15</td>
<td>496; 482.5; 673;33;4;292;76;</td>
</tr>
<tr>
<td></td>
<td>496</td>
<td>482.5; 136;15;147</td>
</tr>
</tbody>
</table>
Conclusions

- Forgoing pre-training seems to work
- Using intermediate representations as inputs for function search seems promising
- MIPS still an issue
- Not good enough for zero-shot architecture search
- Work to do on several areas:
  - In-depth understanding of what functions the model struggles with
  - Adoption and development of pre-filtering approaches
  - Integration with other data sources such as decompiled code
Questions
References


References (cont.)

