Conference Paper Title*

*Note: Sub-titles are not captured in Xplore and should not be used

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

1. Introduction

```
 {
   "input_parameter_count": 1,
   "input_parameter_types": [
     "Pointer"
   ],
-  "return_value_type": "None",
+  "return_value_type": "Integer",
   "dominant_operation_categories": [
     "ConditionalBranching",
     "SubroutineCall"
   ],
   "loop_indicators": false,
   "distinct_subroutine_call_targets": 2,
   "use_indexed_addressing_modes": false,
   "notable_integer_constants": [
-    "0x39"
+    "0x39",
+    "0x4"
   ],
   "notable_floating_point_constants": [],
   "count_of_distinct_immediate_values": 3,
   "modifies_input_parameters": false,
   "modifies_global_state": false,
   "memory_allocation_deallocation": false,
   "io_operations": false,
   "block_memory_operations": false,
-  "inferred_algorithm": "Undetermined"
+  "inferred_algorithm": "Initialization"
}
```

Modern software is increasingly reliant on external libraries. For security researchers, detecting whether a binary uses some vulnerable library function is crucial to assess and mitigate vulnerability exposure [1, 2]. For reverse engineers, reducing the amount of repetitive assembly functions to analyze means being more productive, and allows them to focus on the custom parts of a binary. Binary code similarity detection (BCSD) is a binary analysis task that tries to solve these problems, by determining whether two compiled code fragments behave in similar ways. This task is becoming increasingly important as the size, modularity and rate of production of software grows. For instance, when libraries are statically linked to a binary, BCSD can help to quickly identify which functions are part of common external libraries and which ones have never been seen before. Furthermore, if a vulnerability is found in a library, BCSD can be used to efficiently determine whether a proprietary binary or firmware is using the vulnerable library.

Early approaches to BCSD used human defined heuristics to extract a ”feature vector” from a binary function. These heuristics could be calculated by statically examining the functions and its control flow graph (CFG), or could be measured at runtime by executing the function in an emulated environment. These methods were deterministic and had the benefit of producing human understandable feature vectors, but were often too simplistic or sometimes used computationally intractable alogrithms in the case of CFG analysis.

More recently, machine learning (ML) based methods have shown to be better performing. These methods work by producing an embedding for a binary function using techniques coming from natural language processing. The generated embeddings are usually floating point tensors and serve as the ”feature vector”. These vectors are compared with each other by using metrics such as cosine similarity.

Our work presents a method to effectively find binary clones across binaries using any pre-trained large language model (LLM). The method is simpler than other ML approaches, requires no training nor fine-tuning, and matches state-of-the-art results. It has the unique advantage of generating human interpretable feature vectors instead of numerical values. Additionally, it effectively scales with the performance and size of the LLM used, and thus benefits from the ample amount of research on in this area.

Identify applicable funding agency here. If none, delete this.

1.1. Contributions

- We provide an elementary approach to BCSD purely based on the recent advancements in large language models (LLM) that is effective at cross-optimization, cross-architecture, and cross-obfuscation retrieval. This method requires no pre-training, and generates human interpretable feature sets.
- We show that our approach scales with the performance and size of the LLM used, and benefits from new methods such as "reasoning" models. (This is great since significant investments and research goes to LLMs, blah blah)
- We conduct experiments to demonstrate the capability of our technique and show that it performs as well as other state-of-the-art methods that require pre-training.

Instead of generating a numerical feature vector for a given assembly function, we generate a feature set containing human readable elements.

2. Background

Security researchers and reverse engineers are routinely tasked with the analysis of unknown or proprietary executables. Reverse engineers try to analyze the binary to understand its underlying algorithms, while security researchers want to assess the risk associated with potential vulnerabilities within the executable. This process is usually conducted using a software reverse engineering platform such as Ghidra [24] or IDA Pro [23]. The main functions of these programs are to disassemble and decompile a provided binary, so that its content can be analyzed by humans. Disassembly is the process of retrieving the human-readable assembly instructions from the binary executable, whereas decompilation is the process of generating higher-level pseudo-code from the instructions based on common patterns and heuristics.

BCSD is the process of determining whether two fragments of binary code perform similar actions. These fragments are usually first disassembled, and are then compared for similarity. In practice, similarity detection is performed with one known fragment (either because it was analyzed before or because its source code is known), and one unknown fragment. If the unknown piece of code is deemed highly similar to the known one, it greatly simplifies the analysis task, and reduces duplicate work. Known code fragments are typically collected in a database which is queried against for clone search.

Recent research uses deep learning to generate a vector embedding for each assembly function [11, 14, 16, 17, 19]. Generally, training a model to perform such task requires a large training dataset, and specialized hardware to perform a lot of computations. Once trained however, these methods can generate excellent results. State of the art implementations are limited by their training data, and poorly generalize to out-of-domain tasks, such as code for a different architecture, or code compiled with a different compiler [1, 19]. Most methods also require a pre-processing step after disassembly, such as extracting the control flow graph [16, 17], or tokenizing the input assembly in a specific manner [14, 19].

Production scale BCSD engines contain millions of assembly functions. With one vector embedding per function, nearest neighbor search takes a significant amount of time as the algorithm has to linearly compare the query with every function in the database. To alliviate this issue, approximate nearest neighbor search is used on large databases to reduce the time complexity of the search [21, 22], but it may not always return the first match and thus skews similarity scores.

3. Method

Our method is designed to solve some of the pain points associated with state of the art deep learning models for BCSD. An important factor is that our method extracts human understandable features from an assembly function, instead of a vector embedding. This unique advantage allows immediate human verification when a match is detect by the similarity engine. A database of assembly functions and their interpretable features is more easily maintained, as defects can be patched by humans, rather than having to regenerate the whole database when the model is modified.

By using any open-source or commercially available LLM, we entirely sidestep model training by leveraging the extenive and diverse datasets that LLMs are pre-trained on. Our method can be tuned by modifying the instructions provided to the LLM, which is significantly simpler than having to retrain the model and regenerate embeddings for the whole database. The underlying LLM can also replaced seamlessly, meaning that our method scales with the performance improvements of LLMs - an area which is showing impressive growth and development.

Another key advantage of our method also stems from its textual representation of the extracted feature set. As highlighted previously, vector embeddings are computationally expensive to match against in large databases. Textual search is much more scalable than nearest neighbor search, as is evident in modern search engines being able to filter through billions of documents in a fraction of a second.

3.1. Prompt

The first method consists of querying a large language model with a prompt crafted to extract the high-level behavioral features of the provided assembly code fragment. The assembly code fragment does not require preprocessing. As output, the LLM generates a JSON structure containing the extracted features.

TODO: - What type of output do we expect (main different between prompts)? - How do we solve repetition. - How do we ensure a valid json output?

3.1.1. Framing and Conditioning. We use a prelude that contains general information about the task at hand, and the expected response. Recent commercial LLMs have the ability to generate responses following a specified JSON schema. We do not make use of this capability when evaluating commercial models so that the results can be compared to local LLMs, that do not benefit from this option.

¿ You are an expert assembly code analyst, specializing in high-level semantic description and feature extraction for comparative analysis. Your goal is to analyze an assembly routine from an **unspecified architecture and compiler** and provide its extracted high-level features, formatted as a JSON object. For the provided assembly routine, extract the following features and infer the algorithm. Your output **MUST** be a JSON object conforming to the structure defined by these features.

3.1.2. Type Signature. The first feature category extracted is the type signature of the provided assembly function. We only make the distinction between two types: Integer and Pointer. Inferring more information than these two primitive types has shown to be too complicated for current LLMs. We extract the number of input arguments and their types, and also extract the return value type, if any.

3.1.3. Core Logic and Operations. This section specifies what to extract from the function in terms of its logical behavior, and how to determine the kind of operation that the assembly function performs. We list some of the operations extracted here.

- Indication of loops. This is determined by the presence of jump instructions that point back to a previous instruction after some conditions have been checked.
- Indication of jump tables. Evaluated by patterns suggesting calculated jump addresses based on indices, or a series of conditional jumps.
- Extensive use of indexed addressing modes.
- Use of SIMD instructions and registers.
- Number of distinct subroutine call targets.
- Overall logical behavior. Possibilities include: arithmetic operations, bitwise operations, data movement and memory access, and control flow and dispatching operations.

3.1.4. Notable Constants. This section identifies notable integer and floating point constants. These could be common scalar values used by a specific cryptographic algorithm, or the signature bytes used by a file format or protocol. We exclude small values are used as struct offsets, loop counters or stack pointer adjustments.

3.1.5. Side Effects. The prompt also monitors the side effects that the assembly function has on the system. Modification of input arguments is identified when a pointer input is used to write to memory, and modification of global states is detected similarily, when writes to absolute memory addresses or addresses resolved via global data segment pointers occur. Memory allocation and deallocation is determined by the presence of calls to memory management functions like ‘malloc‘ or ‘free‘. Linear memory access patterns are detected by the presence of sequential indexed memory accesses inside loops or across multiple instructions. Finally, system calls and software interrupts are identified by the presence of specific instructions that trigger them.

3.1.6. Categorization. The last section tries to assign a overall category to the assembly function, by basing it on the information collected in the analysis. The final categorization only weakly supports the similarity search because it does not have a large impact on the similarity score. Its purpose is to provide a concise overview for reviewers of the analysis, who might want to understand the function or verify its similarity with the target. Categories include: cryptographic, data processing, control flow and dispatch, initialization, error handling, wrapper/utility, file management, etc.

3.1.7. Examples. To achieve the best results with our method, we utilize few-shot prompting by providing hand crafted examples along with our prompt. In all of our evaluations, three examples are provided, and our ablation study confirms that adding more than three examples provides little to no benefit. Our examples are selected from a diverse source of functions, and are varied in size and architecture to exemplify the space of possibilities in our evaluations.

The prompt by itself is still very performant, and should be acceptable for most applications. A surpising effect of providing examples is that the prompt is no longer needed for the analysis to be effective. Our results show that using enough examples with an empty system promt generates the same results as a standalone system prompt without examples.

3.2. Comparison

ML based methods that generate an embedding for each assembly function generally compare these vectors using numerical methods such as cosine similarity. Since our generated analysis is not numerical, we use an alternative method to compare two assembly functions. We flatten the JSON structure into a dictionary, where booleans, numbers, and strings are the elements, and the concatenated path to those elements is the key. Jaccard similarity (Intesection over union) is used to obtain a similarity measure.

3.3. Dataset

The dataset is composed of 7 binaries: busybox, coreutils, curl, image-magick, openssl, putty, and sqlite3. All were compiled using gcc for the following platforms: x86_64, x86_32, arm, mips, powerpc. For each binary and platform, binary objects were generated for all optimization levels (O0 to O3), stripped of debug symbols. In total, this

yeilds 140 different binaries to analyze. The binaries were dissassembled using IDA Pro, yielding 383658 assembly functions. Functions consisting of less than 3 instructions were not included as part of the dataset. Pairs of equivalent functions from the same platform but distinct optimization level were made for cross optimization evaluation, and pairs from the same optimization level but different platform were formed for cross platform evaluation.

TODO: Table with function count.

3.4. Model

We evaluate both local models of various sizes and commercially deployed models. Qwen2.5-Coder [ref] with sizes 0.5B to 7B is used to run most local evaluations as its small size fitted our GPU capacity. - Qwen3 [ref] with sizes 0.6 to 4B - Gemma-3n [ref] with sizes 0.5B to 4B

Most evaluations and tests were run using Qwen2.5-Coder, and we use this model as a baseline. On all local models, the input context size was limited to 4096 tokens, and output tokens generation to 512. Large assembly functions that did not fit within the input tokens were truncated.

For ablation study, the following commercial models were evaluated on our dataset.

- OpenAI's GPT-4.1-mini [ref] - OpenAI's o4-mini [ref] - Google's gemini-2.5-flash-lite [ref]

4. Experiments

Our experiments are run on a virtual machine with 8 (CPU ID) cpu cores, 100 GB of RAM, and four NVIDIA Quadro RTX 6000 GPUs each having 24GB of RAM. First, we compare our method against other state-of-the-art NLP based approaches to BCSD on our dataset. Second, we run ablation studies on our method, to determine how the size of the model, the number of examples provided, and the different sections of the prompt contribute to our results. Third, we show that embedding models derived from LLMs that are *not* trained on BCSD are on par with BCSD specific embedding models. Finally, we show that the features extracted from our novel method are not properly represented in state-of-the-art embedding methods, and that by combining our method with an embedding model yield significantly better results that state of the art approaches.

4.1. Evaluation Method

The mean reciprocal rank (MRR) and first position recall (Recall@1) metrics are used for evaluation and comparison to other methods. A pool of assembly function pairs is used for evaluation, where both assembly fragments in a pair come from the same source function. For each pair, we compare the generated features for the first element of the pair with all second elements of the pairs contained in the pool. For example, consider a pool of ten pairs (a_i, b_i) for $i \in [1, 10]$, where a_i is compiled for the arm architecture with optimization level 3, and b_i is compiled for the mips architecture with optimization level 3. The features collected for the a_1 is compared for similarity with the features of b_i for $i \in [1, 10]$. A ranking is generated by ordering these comparisons from most to least similar. Recall@1 is successful if b_1 is ranked first, and the reciprocal rank is $\frac{1}{\text{rank}(b_1)}$.

4.2. Comparison with Baselines

Cross optimization retrieval results are presented in table [T1]. We use the MRR (mean recprocal rank) and Recall@1 metrics, which are used by other methods [refs]. Table [T2] contains the same metrics and baselines, but for cross platform retrieval.

T1 desc: Evaluation of the methods with a pool size of 1000. All functions are compiled for the arm architecture using gcc with the optimization levels specified for each column. We use the gemini-2.5-flash model for the best trade-off between efficiency and performance.

5. Related Work

5.1. Static Analysis

Traditional methods make use of static analysis to detect clone assembly routines. With these methods, a trade-off has to be made between the robustness to obfuscation and architecture differences, and the performance of the algorithm. [1] Control flow graph analysis and comparison [3, 4] is known to be robust to syntactic differences, but often involves computationally intractable problems. Other algorithms that use heuristics such as instruction frequency, longest-common-subsequence, or hashes [5, 6, 7] are more efficient, but tend to fixate on the syntactic elements and their ordering rather than the semantics.

5.2. Dynamic Analysis

Dynamic analysis consists of analyzing the features of a binary or code fragment by monitoring its runtime behavior. For BCSD this method is compute intensive and requires a cross-platform emulator, but completely sidesteps the syntactic aspects of binary code and solely analyzes its semantics. [2] As such, this method is highly resilient to obfuscations, but requires a sandboxed environment and is hard to generalize across architectures and application binary interfaces [ref].

5.3. Machine Learning Methods

TODO: Missing specification of which is platform specific and requires pre-training

The surge of interest and applications for machine learning in recent years has also affected BCSD. Most state-of-the-art methods use natural language processing (NLP) to achieve their results [refs]. Notably, recent machine learning

approaches try to incorporate the transformer architecture into BCSD tasks [refs].

Asm2Vec [17] is one of the first methods to use a NLP based approach to tackle the BCSD problem. It interprets an assembly function as a set of instruction sequences, where each instruction sequence is a possible execution path of the function. It samples these sequences by randomly traversing the CFG of the assembly function, and then uses a technique based on the PV-DM model [18] to generate an embedding for the assembly function. This solution is not cross architecture compatible and requires pre-training.

SAFE [11] uses a process similar to Asm2Vec. It first encodes each instruction of an assembly function into a vector, using the word2vec model [12]. Using a Self-Attentive Neural Network [13], SAFE then converts the sequence of instruction vectors from the assembly function into a single vector embedding for the function. Much like Asm2Vec, SAFE requires pre-training, but can perform cross architecture similarity detection.

Order Matters [16] applies a BERT language reprensentation model [15] along with control flow graph analysis to perform BCSD. It uses BERT to learn the embeddings of instructions and basic blocks from the function, passes the CFG through a graph neural network to obtain a graph semantic embedding, and sends the adjacency matrix of the CFG through a convolutional neural network to compute a graph order embedding. These embeddings are then combined using a multi-layer perceptron, obtaining the assembly function's embedding.

A more recent BCSD model, PalmTree [14], also bases its work on the BERT model [15]. It considers each instruction as a sentence, and decomposes it into basic tokens. The model is trained on three tasks. 1. As is common for BERT models, PalmTree is trained on masked language modeling. 2. PalmTree is trained on context window prediction, that is predicting whether two instructions are found in the same context window of an assembly function. 3. The model is also trained on Def-Use Prediction - predicting whether there is a definition-usage relation between both instructions.

The model CLAP [19] uses the RoBERTa [20] base model, to perform assembly function encoding. It is adapted for assembly instruction tokenization, and directly generates an embedding for a whole assembly function. It is also accompanied with a text encoder (CLAP-text), so that classification can be performed using human readable classes. Categories or labels are encoded with the text encoder, and the assembly function with the assembly encoder. The generated embeddings can be compared with cosine similarity to calculate whether the assembly function is closely related or opposed to the category. This model requires pre-training and is architecture specific (x86_64 compiled with GCC).

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.