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

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 code fragment. 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 a floating point tensor embedding for a code fragment, often using techniques from natural language processing. The generated embeddings serve as the feature vector, which are compared with each other using metrics such as cosine similarity. when filtering through large databases, nearest neighbor search or an approximated variant is used to find the most similar code fragments.

Our work presents a method to effectively find assembly fragment clones across binaries using any available 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 advantage of generating human interpretable feature vectors like earlier approaches, instead of numerical embeddings. 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.

TODO: expand on how our method differs from and compares to others.

1.1. Contributions

- We provide an elementary approach to BCSD purely based on the recent advancements in large language models that is effective at cross-optimization and cross-architecture retrieval. This method requires no pre-training, and generates human interpretable feature vectors.
- We show that our approach scales with the performance and size of the LLM used, and out performs state-of-the-art BCSD models both in versatility and raw metrics.
- We develop a method to combine our model we any pre-existing or future assembly function embedding model, and show that this combination generates the best results.
- We outline ideas for future research to improve the efficiency and performance of this method, and explain how this method could be implemented in production environments.

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

```
loc_42c1c0:
    push rbp
    mov rbp, rsp
    sub rsp, 8
    mov [rbp+var_8], rdi
    mov rax, [rbp+var_8]
    mov rdi, rax
    call sub_42b49a
    mov rax, [rbp+var_8]
    mov dword ptr [rax+50h], 0
    mov rax, [rbp+var_8]
    mov dword ptr [rax+58h], 0
    mov rax, [rbp+var_8]
    mov edx, [rax+58h]
    mov rax, [rbp+var_8]
    mov [rax+54h], edx
    nop
    leave
    retn
```

```
loc_41ac99:
    mov dword ptr [rdi], 67452301h
    mov dword ptr [rdi+4], 0efcdab89h
    mov dword ptr [rdi+8], 98badcfeh
    mov dword ptr [rdi+0ch], 10325476h
    mov dword ptr [rdi+50h], 0
    mov dword ptr [rdi+58h], 0
    mov dword ptr [rdi+54h], 0
    retn
```

Figure 1: The `MD5Init` function from `putty`, compiled with gcc for the `x86_64` architecture using optimization levels 0 and 3 respectively. Our method is able to identify the fragments as clones.

2. Background

2.1. Binary analysis

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 found within the executable. This process is usually conducted using a software reverse engineering platform such as Ghidra [ref] or IDA Pro [ref]. 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.

Binary analysis is a hard task because once a program is compiled, most of the information contained in its source code is lost [2]. Variables, data structures, functions, and comments are removed, because the compiler's task is to make the program as efficient as possible—which often means removing as much as possible. The optimizers within the compiler only have a single rule: They must not make changes to the observable behavior of the program (often refered to as the "as-if rule" [ref]). As a result, compilers can remove, reorder, and inline significant parts of the code, making it difficult to understand its behavior. Even worse, adverserial programs such as malware or digital rights management software make use of obfuscation techniques to resist having their code reverse engineered.

2.2. Binary code similarity detection

BCSD is the task 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. For example, if a major vulnerability in a widely used, core open-source component is found, BCSD can be used to quickly assess if a binary contains the vulnerable code fragment. It can also be used for plagiarism detection, which could take the form of license or patent infringement, for software and malware classification [3], or for security patch analysis [4]. Challenges in binary analysis are amplified in binary code similarity detection, because two code fragments that seem very different can still have the same observable behavior.

Recent research uses deep learning to generate a vector embedding for each assembly function [5]–[9]. Generally, training a model to perform such task requires a large training dataset, and highly performant GPUs. 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 assembly code for a different architecture, or code compiled with a different compiler [2], [9]. Most methods also require a pre-processing step after disassembly, such as extracting the control flow graph [7], [8], or slightly modifying the input assembly in a specific manner [6], [9].

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 compare the query with every function in the database in linear time complexity. To alliviate this issue, approximate nearest neighbor search is used on large databases to reduce the time complexity of the search [10], [11], but it may not always return the first match and thus skews similarity scores.

2.3. Problem definition

We deem two assembly code fragments to be semantically identical if they have the same observable behavior on a system. This type of clone is generally referred to as type 4 clones [8], [12], and is considered the most difficult to detect. In research, it is common to use the same source code function compiled with different compilers or compilation options to create such type four clones [ref]. This type of clone excludes procedures that happen to have the same output but achieve it in different ways, such as the breadth-first search and depth-first search algorithms. Instead of generating a binary output as to whether two code fragments are clones, the feature vector generated for each assembly function are compared to generate a similarity index. This index is a value between 0 and 1 that represents the degree of similarity between the two code fragments. This more closely aligns with real world use cases where two pieces of code are highly similar but not identical, such as a vulnerable function and its patched version.

TODO: more?

3. Method

Our method is designed to address some of the pain points of 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 detected 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 without invalidating the database, meaning that our method will continue to scale with the performance improvements of LLMs—an area which is showing impressive growth and development. Furthermore, if a section of the prompt was edited to modify the output feature set, the database can still be maintained without having to regenerate an output for each item. Default values or values derived from other fields in the feature set can be added, as is standard with database migrations.

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 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. We outline the prompt made up of multiple parts, each designed to extract specific semantic information from the assembly function. The full prompt is open-source and avaliable on Github.

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.

Local LLMs, especially the smallest models, will sometimes generate nonsensical output. In our early experiments, the smallest local model evaluated (0.5B parameters) would sometimes repeat the same line of JSON until it ran out of context space, or generated invalid JSON. To combat these artifacts, we run inference again when JSON parsing fails and increase the temperature of output token selection. This is done in a loop until valid JSON is generated. In our experiments, the maximal amount of trials required for any query to generate valid json was three, but 95% of the generated responses from the smallest model would constitute valid JSON without retries.

> 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. 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.

TODO: List style would be better.

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.

```
 {
   "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"
 }
```

Figure 2: TODO

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.

TODO: one or two more sentences to explain, seems vague rn.

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, 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 112 different binaries to analyze. The binaries were dissassembled using IDA Pro and separated into individual

functions, yielding 383 658 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

TODO: This section

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 ML 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. Finally, we demonstrate 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 yields 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 [refs]. 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 generated for function 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 found to be the most similar function, and the reciprocal rank metric is calculated as follows.

$$\frac{1}{\text{rank}(b_1)}$$

4.2. Clone search with different optimization levels

This experiment benchmarks the capability of the baselines and our method for detection of similar code fragments across different optimization levels. As evident in Table 1, the hardest retrieval task is between optimization levels 0 and 3, because there is a substantial difference between code compiled with `-O0` and code compiled with `-O3` (figure 1). At optimization level 0, functions perform a lot of unessecary actions, such as extensively moving data between registers and perform conditional evaluation of expressions that return a constant value. The generated code mostly is mostly left untouched by the optimizer. At optimization level 3, the compiler will inline simple functions into the body of the caller, meaning that jumps and calls to other places in the binary are replaced by the destination's instructions. Some loops are unrolled, so that each iteration of the loop is laid out sequentially instead of performing a conditional check and a jump to the loop's initial instruction. Also, instructions can be heavily reordered to achieve best performance, while keeping the observable behavior of the program untouched.

The baselines are SAFE, OrderMatters, PalmTree, Asm2Vec, and CLAP. A summary of each of their architecture is provided in the Related Work section (TODO: link?). We also include Qwen3-Embedding 4B, an embedding model based on Qwen3 trained for general text embedding as baseline. We present the results of our method evaluated on both a local model, Qwen2.5-Coder 7B Parameters, and a commercially deployed model, Gemini 2.5 Flash.

The baselines mostly perform worse than expected on this evaluation. As our own dataset is used rather than one that was presented by the baselines, we believe this may be caused by overfitting in the training process of these models. As evident here, one of our method's advantage is that it requires no fine-tuning to acheive good results, and thus should generalize well to unseen settings. Qwen3-Embedding 4B also generates impressive results, given that it was not trained for assembly clone detection.

4.3. Clone search with different architectures

Different CPU architectures have varying assembly code languages. It is hard for BCSD methods that analyze assembly code to support multiple architectures. These methods need to accurately represent two functions with completely different syntaxes but with identical semantics as being very similar in terms of their feature vector or embedding. Hence, methods that use CFG analysis have a better chance at supporting many architectures, since the structure of the CFG itself is architecture agnostic. However, the basic blocks

Model	MRR						Recall @ 1					
	O0-O1	O0-O2	O0-O3	O1-O3	O2-O3	average	O0-O1	O0-O2	O0-O3	O1-O3	O2-O3	average
Order Matters	0.006	0.008	0.006	0.006	0.006	0.006	0.001	0.002	0.001	0.000	0.001	0.001
SAFE	0.189	0.200	0.189	0.218	0.171	0.193	0.000	0.063	0.063	0.063	0.000	0.038
PalmTree	0.020	0.019	0.230	0.314	**0.878**	0.292	0.006	0.007	0.080	0.184	0.676	0.191
Asm2Vec	0.494	0.460	0.444	0.535	0.563	0.499	0.290	0.252	0.234	0.343	0.376	0.299
CLAP	0.244	0.221	0.214	0.550	0.781	0.402	0.187	0.176	0.168	0.455	0.707	0.339
Qweni 2.5 7B	0.471	0.412	0.343	0.456	0.608	0.458	0.342	0.301	0.234	0.345	0.488	0.342
Gemini 2.5 Flash	**0.739**	**0.672**	**0.568**	**0.700**	0.816	**0.699**	**0.646**	**0.579**	**0.485**	**0.618**	**0.758**	**0.617**

TABLE 1: Evaluation of the baselines and our method on cross optimization retrieval with a pool size of 1000. All functions are compiled for the arm architecture using gcc with the optimization levels specified for each column. Three examples are provided with our prompt.

Model	MRR				Recall @ 1			
	arm-x86_64	powerpc-x86_64	mips-x86_64	average	arm-x86_64	powerpc-x86_64	mips-x86_64	average
SAFE	0.239	0.187	0.196	0.207	0.063	0.063	0.063	0.063
PalmTree	0.037	0.036	0.018	0.030	0.031	0.013	0.007	0.017
Asm2Vec	0.242	0.293	0.417	0.317	0.085	0.113	0.231	0.143
Order Matters	0.007	0.007	0.007	0.007	0.002	0.000	0.001	0.001
CLAP	0.416	**0.523**	0.494	0.478	0.334	**0.443**	0.415	0.397
Qwen 2.5 7B	0.263	0.201	0.202	0.222	0.165	0.108	0.110	0.128
Gemini 2.5 Flash	**0.548**	0.520	**0.525**	**0.531**	**0.436**	0.414	**0.417**	**0.422**

TABLE 2: Evaluation of the baselines and our method on cross architecture retrieval with a pool size of 1000. All functions are compiled with optimization level 2 using `gcc` with the architecture specified for each column. The same baselines and models are used as in the cross architecture evaluation. Three examples are provided with our prompt.

that constitute this graph are still in assembly code, which does not fully resolve the issue. Furthermore, there exists many different variants of each instruction set, because each new version of an architecture brings new instructions to understand and support. With deep learning methods, this requires training or fine-tuning the model to understand a language variant every time. Afterwards, all embeddings in a BCSD database need to be regenerated. Our method does not directly address this issue, but brings a significant improvement. It indirectly makes use of the vast amount of data used to train foundational LLMs. Since a LLM has extensively seen all of the mainstream CPU architectures and their dialects in its training data, it is able to grasp their meaning and extract features from them. If the model in use seems to poorly comprehend a specific architecture, it can be replaced with one that better performs the specific platform without invalidating the BCSD database.

Our method surpasses the baselines, but more work in this area is clearly still needed. The recall@1 metrics show that the best method is able to rank the the correct assembly fragment in first place only 42% of the time on average.

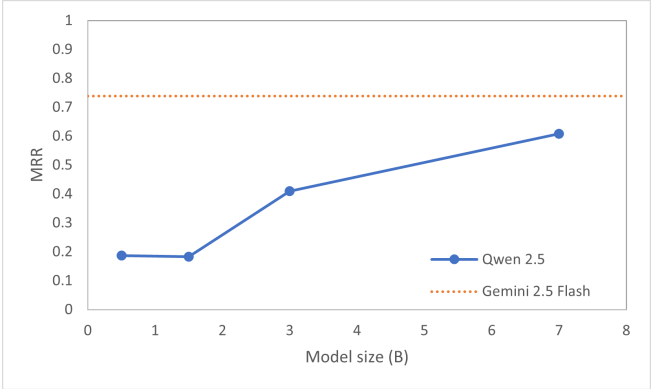

Figure 3: MRR performance for cross optimization retrieval against the number of parameters in the LLM. All assembly functions are compiled with `gcc` for the arm architecture. Retrieval is performed between optimization levels 0 and 1 with a pool of 1000 assembly functions. Three examples are provided with our prompt.

4.4. Ablation on model size

In this experiment, we vary the LLM size to determine the correlation between the number of parameters in the LLM and the performance of our method on BCSD retrieval. Our results generally follow the scaling laws for neural language models [13], in that increasing the model size does significantly improve the results generated.

From our observations, LLMs with less than 3B parameters do not seem to comprehend the analysis task when they are not provided with any examples. When provided with examples, these small models will mimic the examples provided without basing the output on the assembly function in the query. We can clearly see a form of sub-linear increase

in performance with respect to model size. The Gemini 2.5 Flash model architecture is not disclosed at the time of writing, but we can expect the model to have an order of magnitude more parameters, than our local models, and may use a mixture of expert architecture, based on disclosed previous Gemini model versions.

TODO: embedding model size ablation?

4.5. Ablation on examples

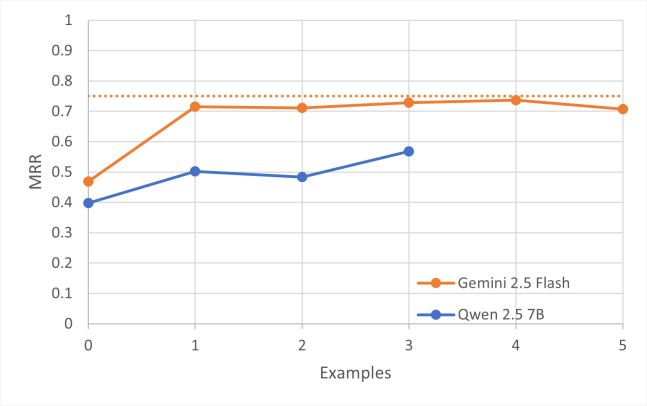

Figure 4: MRR performance for cross optimization retrieval against the number of examples provided in the prompt. Functions are compiled using gcc for the x86-64 architecture. A pool of 100 assembly functions is used, with retrieval between optimization levels 0 and 3 for Gemini 2.5 Flash, and 0 and 1 for Qwen 2.5 7B.

By providing hand crafted examples to the large language model, we are able to increase the performance of the assembly function analysis. This follows the general observations behind few show prompting [14]. Providing a single example significantly increases the retrieval scores, but providing more than one provides very limited increases in scores. Interestingly, a smaller model such as Qwen 2.5 7B still sees marginal increase in MRR scores as the number of examples increases up to 3. This relates to our previous observation that small LLMs rely more on the provided examples than larger models. Another evidence of this is the number of invalid JSON outputs generated by Qwen 2.5 7B. When no examples are provided, approximately 3% of queries generate an invalid output. With only one example, no invalid outputs are generated, and the same applies when 2 or 3 examples are provided.

With our experimental configuration, providing 4 examples to Qwen 2.5 7B significantly decreases its MRR scores. That is because the examples almost completely use the full context window that we provide to the LLM. As such, most query assembly functions are too large and thus truncated, which loses information about our query.

The dotted line in figure 4 represents the MRR score obtained by providing no examples to Gemini 2.5 Flash, but providing a reference JSON schema for the model to follow. The scores being equivalent shows that Gemini does not base itself on the provided examples, but only uses them to understand the JSON schema required. In all our other evaluations, we provide examples instead of a JSON schema because local models do not have the capability of generating output based on a schema built-in.

4.6. Ablation on the prompt used

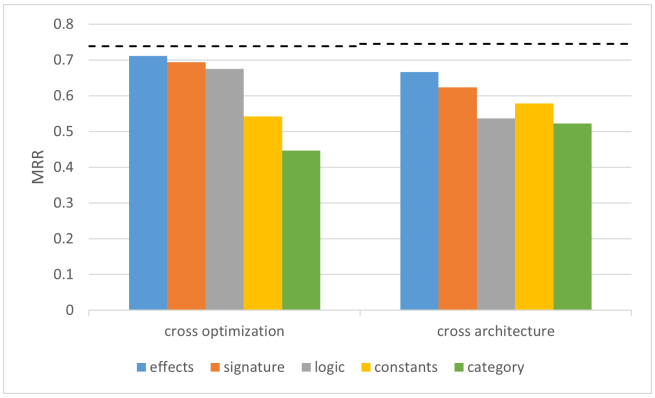

Figure 5: MRR performance for the prompt with one of the sections removed. Cross optimization retrieval is done between fragments at optimization levels 0 and 3, all functions are compiled for the x86-64 architecture. Cross architecture retrieval is done between functions compiled for arm and x86-64, all are compiled using optimization level 2. Both tasks are done with a pool size of 100. The dotted lines show the MRR score for the prompt without any sections removed. A larger gap between the line and bar means the prompt section has more impact.

To verify that each section of our prompt brings meaningful insight into the assembly function analysis task, we perform an ablation study by removing one section of the prompt while keeping others intact. We notice that all individual sections bring a positive outcome to the overall results, but some sections of the prompt have a larger impact than others. In particular, the categorization and notable constants sections have the most impacts on cross optimization retrieval. The categorization section causes an increase of almost 0.3 on the MRR metric, when evaluated on the the hardest cross optimization task. The notable constants sections brings an increase of almost 0.2 on the MRR for the same task. This result is justified by the comparatively large range of accepted values for both features. For example, the list of notable constants has many more possible values and thus has more variability in the output compared to a set of booleans, such as those in the side effects prompt section.

The story is slightly different when this ablation study is performed for cross architecture retrieval instead. As seen on figure 5, there is a smaller difference between the least and most influential prompt sections. The categorization and notable constants sections are significantly less impactful than they were for the cross optimization study, while the other three sections have a larger impact. For instance, the

categorization section went from having an impact of 0.30 on the MRR to having an impact 0.22, while the logic section is more influential, going from 0.06 to 0.20 in MRR impact.

TODO: Links to sections?

4.7. Combined Method

Our method has shown to be an excellent generalist on BCSD retrieval tasks. Nevertheless, models trained specifically for assembly function understanding and BCSD can out-perform our method on functions similar to the dataset they were trained on. Even if our method is more scalable, state-of-the-art embedding learning models are still useful for specific tasks, such as supporting niche architectures or understanding similarity through obfuscated code. The state-of-the-art embedding models are comparatively much smaller than our method. For instance the CLAP [9] model baseline is only 110M parameters, compared to the many billions required to perform our method. Once trained, small embedding models are much faster than LLM inference, but the cost of training such models is still very expensive. Their small size also means that these models can be run on CPU for small pool sizes.

To achieve a best of both world scenario, we experiment with combining the similarity scores from an embedding model and our prompting method. Our evaluation method consists of generating both an embedding and a LLM analysis for each function. The embedding similarity s_e is calculated using cosine similarity, and the analysis similiarity s_a is calculated using jaccard similarity. Both similarity scores are then combined with equal weight.

$$S = \frac{s_e + s_a}{2}$$

To provide a method that keeps some of the advantages of our presented work, we use Qwen3-Embedding 4B as the embedding model for this experiment. As such, the combination still does not require any training or fine-tuning, as we have shown that off-the-shelf embedding models based on a LLM perform very well. Furthermore, using a generic embedding model means that it can inexpensively be replaced by a new generation (this can be done without having to train a new model).

The combined method significantly surpasses both the embedding and analysis methods alone. Seen differently, the embeddings and analysis supplement each other, meaning that our analysis extracts features from the assembly function that are not properly represented in the embeddings model.

	Gemini 2.5 Flash	Qwen Embedding	Combined
O0-O1	0.646	0.64	0.91
O0-O2	0.579	0.554	0.843
O0-O3	0.485	0.518	0.759
O1-O3	0.618	0.64	0.855
O2-O3	0.758	0.783	0.921
arm-x86_64	0.436	0.334	0.736
powerpc-x86_64	0.414	0.443	0.746
mips-x86_64	0.417	0.415	0.729

TABLE 3: Comparison between Qwen3-Embedding 4B, Gemini 2.5 Flash, and the combination of both. The retrieval task is performed on both cross optimization and cross architecture settings. For cross optimization, the binaries are compiled for the arm architecture, for cross architecture, the binaries are compiled with optimization level 2. A pool of 1000 assembly functions is used throughout. Only Recall @ 1 scores are presented.

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. [2] Control flow graph analysis and comparison [15], [16] 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 locality sensitive hashes [3], [17], [18] are less time consuming, 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. [1] 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 [19].

5.3. Machine learning methods

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 [8] 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 [20] to generate an

embedding for the assembly function. This solution is not cross architecture compatible and requires pre-training.

SAFE [5] uses a process similar to Asm2Vec. It first encodes each instruction of an assembly function into a vector, using the word2vec model [21]. Using a Self-Attentive Neural Network [22], 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. This method supports both cross optimization and cross architecture retrieval, but was only trained on the 'AMD64' and 'arm' platforms.

Order Matters [7] applies a BERT language reprensentation model [23] 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. This method supports cross architecture and cross platform tasks, although its implementation is only trained on `x86-64` and 'arm' for cross platform retrieval.

A more recent BCSD model, PalmTree [6], also bases its work on the BERT model [23]. 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. This method's reference implementation is only trained on cross compiler similarity detection, but can be trained for other tasks.

The model CLAP [9] uses the RoBERTa [24] 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 training and is architecture specific (`x86-64` compiled with `gcc`).

6. Conclusion

Our method for BCSD introduces a shift in perspective compared to previous state-of-the-art BCSD embedding models. However, this new approach maintains some of the known limitations and also brings new limitations. Most importantly, this method makes use of massive models compared to previous methods. A powerful set of GPUs is required when generating feature sets for a large pool or database. This favors centralised databases with large amounts of assembly fragments over locally maintained databases with hundreds or thousands of functions. Otherwise, a commercially deployed LLM can be used at a cost, but concerns surrounding data privacy can legitimately be raised. Another potential limitation is the size of the feature set extracted during assembly analysis. As shown, our prompt performs very well with pool sizes of 1000 assembly functions, however, this may not be the case when comparing millions of feature sets, as is the case in production databases.

We remain hopeful that our method can bring significant improvements to the current landscape of BCSD, since it resolves many of the previously acknowledged limitations. Our method does not require any form of training, and can be performed with any LLM available. It also offers a distinct advantage over the current state-of-the-art, because the feature vectors generated are human interpretable. This makes the method easily tunable by human experts, the similarity scores are much more easily verifiable, and cases of incorrect similarity detection can be explained using the generated feature set. Furthermore this method can be scaled to databases containing millions of assembly functions compared to embedding models, since inverted index search has a lower time complexity than nearest neighbor search or its approximated versions.

6.1. Future research

Our results open avenues for further investigation in LLM based BCSD, and more broadly in LLM assisted reverse engineering. A few of these oportunities are outlined here.

6.1.1. Reasoning model. Newest state of the art LLMs are fine tuned to natively use chain of thought methods [25] before answering the query. These models are known as reasoning models, and it has been shown that this method of inference produces more accurate and higher quality responses [25]–[27]. It may be worth evaluating whether this style of model performs better than non reasoning models (as used in our research) on assembly function analysis.

6.1.2. Fine-tuning. As stated, our method does not require fine-tuning, and was not evaluated on fine-tuned model. Developing a fine-tuning step for assembly code understanding and analysis may prove to be efficient and bring significant improvements to our method. An example of such technique would be distillation. With distillation, a LLM is used to fine-tune a much smaller language model. Since there is no ground truth in text generation, the distillation process uses the LLM output as ground truth to train the smaller model. As we have shown, a large model such as Gemini 2.5 Flash performs remarkably better than a small model such as Qwen 2.5 7B. We believe the distillation process could considerably reduce this gap.

TODO: Agent stuff, what else?

References

[1] Z. Liu, "Binary code similarity detection," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1056–1060.

[2] L. Ruan, Q. Xu, S. Zhu, X. Huang, and X. Lin, "A survey of binary code similarity detection techniques," *Electronics*, vol. 13, no. 9, 2024. [Online]. Available: https://www.mdpi.com/2079-9292/13/9/1715

[3] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: opcode-sequence-based malware detection," in *Proceedings of the Second International Conference on Engineering Secure Software and Systems*, ser. ESSoS'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 35–43. [Online]. Available: https://doi.org/10.1007/978-3-642-11747-3_3

[4] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: Security patch analysis for binaries towards understanding the pain and pills," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 462–472.

[5] L. Massarelli, G. A. D. Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Safe: Self-attentive function embeddings for binary similarity," 2019. [Online]. Available: https://arxiv.org/abs/1811.05296

[6] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3236–3251. [Online]. Available: https://doi.org/10.1145/3460120.3484587

[7] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, pp. 1145–1152, Apr. 2020. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5466

[8] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 472–489.

[9] H. Wang, Z. Gao, C. Zhang, Z. Sha, M. Sun, Y. Zhou, W. Zhu, W. Sun, H. Qiu, and X. Xiao, "Clap: Learning transferable binary code representations with natural language supervision," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 503–515. [Online]. Available: https://doi.org/10.1145/3650212.3652145

[10] A. Andoni, P. Indyk, and I. Razenshteyn, "Approximate nearest neighbor search in high dimensions," 2018. [Online]. Available: https://arxiv.org/abs/1806.09823

[11] P. Indyk and H. Xu, "Worst-case performance of popular approximate nearest neighbor search implementations: Guarantees and limitations," 2023. [Online]. Available: https://arxiv.org/abs/2310.19126

[12] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi, "Binclone: Detecting code clones in malware," in *2014 Eighth International Conference on Software Security and Reliability (SERE)*, 2014, pp. 78–87.

[13] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020. [Online]. Available: https://arxiv.org/abs/2001.08361

[14] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[15] T. Dullien, "Graph-based comparison of executable objects," 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:2001486

[16] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 480–491. [Online]. Available: https://doi.org/10.1145/2976749.2978370

[17] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling, "Detecting clones across microsoft .net programming languages," in *2012 19th Working Conference on Reverse Engineering*, 2012, pp. 405–414.

[18] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary function clustering using semantic hashes," in *2012 11th International Conference on Machine Learning and Applications*, vol. 1, 2012, pp. 386–391.

[19] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: dynamic similarity testing for program binaries and components," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 303–317.

[20] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," 2014. [Online]. Available: https://arxiv.org/abs/1405.4053

[21] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013. [Online]. Available: https://arxiv.org/abs/1310.4546

[22] Z. Lin, M. Feng, C. N. dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A structured self-attentive sentence embedding," 2017. [Online]. Available: https://arxiv.org/abs/1703.03130

[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019. [Online]. Available: https://arxiv.org/abs/1810.04805

[24] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019. [Online]. Available: https://arxiv.org/abs/1907.11692

[25] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023. [Online]. Available: https://arxiv.org/abs/2201.11903

[26] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," 2023. [Online]. Available: https://arxiv.org/abs/2205.11916

[27] T. Wu, J. Lan, W. Yuan, J. Jiao, J. Weston, and S. Sukhbaatar, "Thinking llms: General instruction following with thought generation," 2024. [Online]. Available: https://arxiv.org/abs/2410.10630

[28] S. H. Ding, B. C. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 461–470. [Online]. Available: https://doi.org/10.1145/2939672.2939719