

Reasoning Like a Reverse Engineer: Language-Model as Feature Extraction for Binary Similarity

Anonymous submission

Abstract

Binary code similarity detection is a core task in reverse engineering. It supports malware analysis and vulnerability discovery by identifying semantically similar code in different contexts. Modern methods have progressed from manually engineered features to vector representations. Hand-crafted statistics (e.g., operation ratios) are interpretable, but shallow and fail to generalize. Embedding-based methods overcome this by learning robust cross-setting representations, but these representations are opaque vectors that prevent verification. They also face a scalability–accuracy trade-off, since high-dimensional nearest-neighbor search requires approximations that reduce precision. Current approaches thus force a compromise between interpretability, generalizability, and scalability.

We bridge these gaps using a language model-based agent to generate structured analyses of assembly code, with features such as input/output types, side effects, notable constants, and algorithmic intent. Unlike hand-crafted features, they are richer and adaptive. Unlike embeddings, they are human-readable, maintainable, and directly searchable with inverted or relational indexes. Without any matching training, our method respectively achieves 42% and 62% for recall@1 in cross-architecture and cross-optimization tasks, comparable to embedding methods with training (39% and 34%). Combined with embeddings, it significantly outperforms the state of the art, demonstrating that accuracy, scalability, and interpretability can coexist.

1 Introduction

Modern software is increasingly reliant on external libraries. For security researchers, detecting whether some executable uses a vulnerable library function is crucial to assess and mitigate vulnerability exposure [31, 36]. 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 that vulnerable library.

TODO: Explain why the problem is so hard.

Early approaches to BCSD used human defined heuristics to extract a “feature vector” from a binary code fragment [2, 12, 37]. 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 [37] or sometimes used computationally intractable algorithms [12], in the case of CFG analysis.

More recently, machine learning (ML) based methods have shown to be better performing [11, 28, 33, 45, 50]. These methods produce a floating point vector embedding for each code fragment, often using techniques from natural language processing. The generated embeddings serve as the feature vector, which are compared with each other using vector distance 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 code 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 surpasses 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 in

this area.

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 training other than the pre-training step already completed by LLM developers, and generates human interpretable feature vectors.
- We show that our approach scales with the performance and size of the LLM used, and outperforms state-of-the-art BCSD models both in versatility and raw metrics. We also present
- We develop a method to combine our model with any pre-existing or future code embedding model, and show that this combination can generate outstanding results, significantly outperforming state-of-the-art embedding models.
- We outline ideas for future research to improve the efficiency and performance of this method, and explain how it could be implemented in production environments.

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 [1] or IDA Pro [19]. The main functions of these platforms are to disassemble and decompile the provided machine code, so that its content can be analyzed by humans. Disassembly is the process of retrieving the assembly instructions (human-readable machine code) 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 [36]. 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 referred to as the “as-if rule” [23]). As a result, compilers can remove, reorder,

and inline significant parts of the code, making it difficult to understand the code’s behavior. Even worse, adversarial 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. Known code fragments are typically collected in a database which is queried against for clone search. If the unknown piece of code is deemed highly similar to the known one, the analysis task becomes significantly simpler and duplicate work is minimized. For example, if a vulnerability in a widely used 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 [37], or for security patch analysis [49]. 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 [11, 28, 33, 45, 50]. 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 [36, 45]. Most methods also require a pre-processing step after disassembly, such as extracting the control flow graph [11, 50], or slightly modifying the input assembly in a specific manner [28, 45].

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 alleviate this issue, approximate nearest neighbor search is used on large databases to reduce the time complexity of the search [3, 22]. This comes at the cost of not always correctly identifying the best match, and thus misrepresents the similarity scores.

2.3 Problem definition

We deem two assembly code fragments to be semantically identical if they have the same observable behavior. This cat-

```

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_43EE04:
addiu $sp, -0x20
sw $ra, 0x18+var_s4($sp)
sw $fp, 0x18+var_s0($sp)
move $fp, $sp
sw $a0, 0x18+arg_0($fp)
lw $v0, 0x18+arg_0($fp)
move $a0, $v0
jal sub_43D930
nop
lw $v0, 0x18+arg_0($fp)
sw $zero, 0x50($v0)
lw $v0, 0x18+arg_0($fp)
sw $zero, 0x58($v0)
lw $v0, 0x18+arg_0($fp)
lw $v1, 0x58($v0)
lw $v0, 0x18+arg_0($fp)
sw $v1, 0x54($v0)
nop
move $sp, $fp
lw $ra, 0x18+var_s4($sp)
lw $fp, 0x18+var_s0($sp)
addiu $sp, 0x20
jr $ra
nop

loc_4241F0:
movdqa xmm0, cs:xmmword_448750
mov dword ptr [rdi+50h], 0
mov dword ptr [rdi+58h], 0
movups xmmword ptr [rdi], xmm0
mov dword ptr [rdi+54h], 0
retn

```

Figure 1: The MD5Init function from putty, compiled with gcc for with different architectures and optimization levels. on the left, compiled for x86-64 with -O0, in the center, compiled for mips with -O0, and on the right, compiled for x86-64 with -O3. Our method is able to identify these functions as clones.

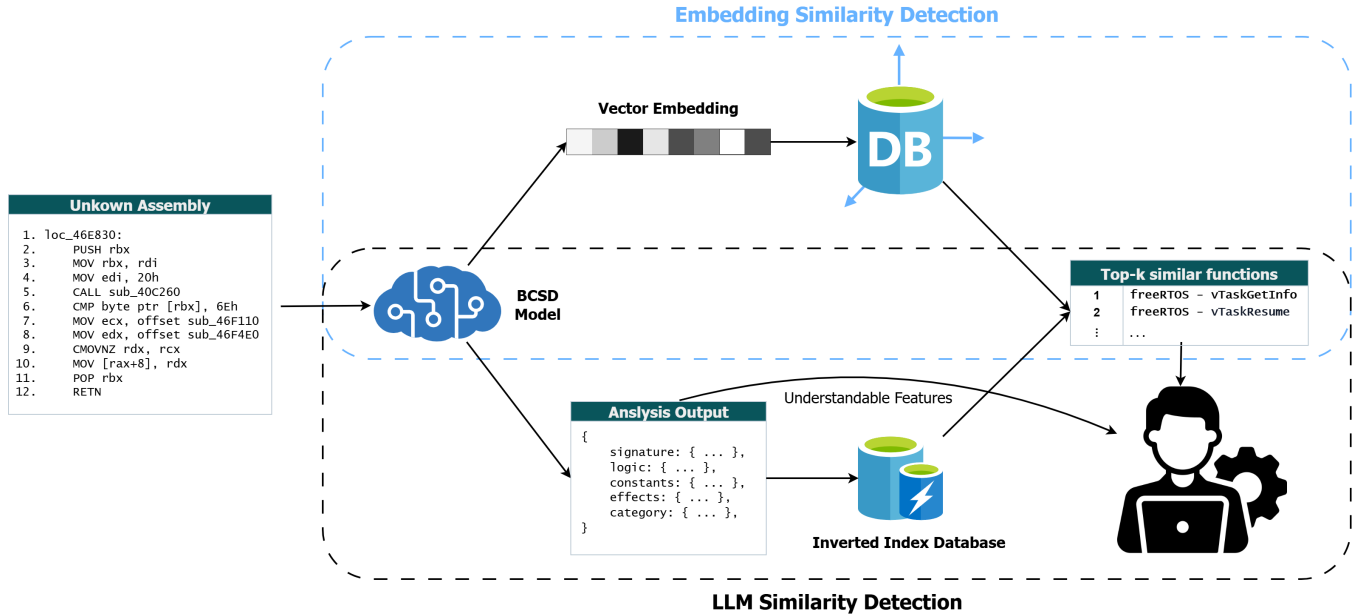


Figure 2: Workflow of our method (bottom) compared to embedding based BCSD (top).

egory of clones is generally referred to as type four clones [11, 15], 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 settings to create such type four clones [11, 28, 33, 45, 50]. 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 score. This value is between 0 and 1 and represents the degree of similarity between the two code fragments. This more closely aligns with real world use cases, where two pieces of code can be highly similar but not identical. For example, two different versions of a function from the same library deemed similar can help understand the nature of an executable, or a vulnerable function and its patched variant matched can help assess that a vulnerability was indeed fixed. That is, the goal is to assign the highest similarity score to type four clones, and lower similarity scores to functions that do not share semantic behavior.

2.4 Motivation

BCSD is a promising technology in the field of software reverse engineering. With an ideal binary similarity engine, the task of analyzing an unknown binary is reduced to analyzing the handful of code fragments that were genuinely developed or modified by the software provider. All functionality coming from shared libraries or known sources, which constitutes a large majority of modern software, would be effectively identified as such and would require no reverse engineering work. Current technology remains far from this scenario. BCSD database maintenance is hard, because embeddings are invalidated every model update. Current methods still exhibit a large portion of false positives, which means that human verification is required. Yet, these techniques give little to no insight as to why a similarity was detected between the assembly fragments.

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 and leverage the extensive and diverse dataset that LLMs are pre-trained on. Our method can be tuned by modifying the instructions provided to the

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 function 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.

Figure 3: Preamble to the analysis task description

language model, which is significantly simpler than having to retrain the model and regenerate embeddings for the whole database. The underlying LLM can also be replaced without invalidating the database, meaning that our method continues to scale with the performance improvements of LLMs—an area which is showing impressive growth and development. Furthermore, if the prompt was edited in a way that alters the output feature set, the database can 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 the textual representation of the extracted feature set. As highlighted previously, vector embeddings are computationally expensive to match against in large databases. Inverted index 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. As output, the LLM generates a JSON structure containing the extracted features. Compared to other methods, the assembly code fragment used in the query does not require any preprocessing. We outline the prompt made up of multiple parts, each designed to extract specific semantic information from the assembly function. The full prompt is available as part of the published artifacts [TODO].

Framing and conditioning. We use a prelude (Figure 3) that contains general information about the task at hand, and the expected response.

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.

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 features 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.pertinence
- The overall logical behavior of the function. Possibilities include: arithmetic operations, bitwise operations, data movement and memory access, and control flow and dispatching operations.

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 that are used as offsets, loop counters or stack pointer adjustments.

Side effects. The prompt also monitors the side effects that the assembly function has on the system. It reports these as a set of booleans values which state whether or not the function has the associated side effect. Each side effect is associated with a heuristic that the language model should use to identify them.

- *Modification of input arguments.* This is mainly identified when a input pointer is used as destination to a memory write.
- *Modification of global state* is detected similarly, when writes to absolute memory addresses or addresses resolved via global data segment pointers occur.
- *Memory management* is determined by the presence of calls to memory allocation and deallocation functions like `malloc` or `free`.
- *Input/Output handling* patterns are detected through characteristic file descriptor management behaviors, including system calls for opening, closing, reading, and writing, as well as instruction sequences implementing buffer mechanisms.

- *System calls and software interrupts* are identified by the presence of the specific instructions that trigger them.

Categorization. The last section tries to assign an overall category to the assembly function, based on the information collected during the analysis. Its purpose is not only to increase the similarity scores when a decisive category is identified, but also to provide a concise overview for analysts who may wish 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.

3.2 Generation Techniques

We make use of a few techniques to ensure that the generated output constitutes valid JSON, and that we maximize the relevance of the generated output.

It is known that model size is one of the biggest factor in model performance, and this is especially true for language model inference [25]. 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 output until it ran out of context space, or generated invalid JSON. To minimize these edge cases, we inference is repeated when JSON parsing fails and the temperature of output token selection is increased. This is performed 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.

To achieve the best results with our method, we also utilize few-shot prompting [5] by providing a few assembly functions accompanied by hand crafted example analyses along with our prompt. In our evaluations against the baselines, three examples are provided, and our ablation study confirms that adding more than three examples provides little to no benefit. Providing a single example or a JSON schema in the prompt is enough to achieve good results. We do not provide a JSON schema because it is redundant information when examples are included. 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.

Recent commercial LLMs have the ability to generate responses following a provided JSON schema that is aside from the input prompt. 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.

3.3 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

Library	Function count	Median function length	Description
BusyBox	66706	32	A fairly complete unix utility set for any small or embedded system.
GNU coreutils	42545	31	The basic utilities of the GNU operating system.
curl	27198	14	Command line tool to transfer data with URLs.
ImageMagick	69873	55	Software suite used for editing and manipulating digital images.
OpenSSL	134359	29	Toolkit for general-purpose cryptography and secure communication.
PuTTY	12655	26	An implementation of SSH and Telnet for Windows and Unix platforms.
SQLite	30322	40	A small SQL database engine.

Table 1: Dataset details.

```

{
  "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,
  "number_of_interrupts_system_calls": 0,
-  "inferred_algorithm": "Undetermined"
+  "inferred_algorithm": "Initialization"
}

```

Figure 4: Analysis comparison of the sha384_init assembly function. Comparison between the analyses of the function compiled for arm (red) and x86-64 (green), with identical values in black. This pair has a similarity score of 0.9.

analysis is not a numerical vector but rather structured text, we use an alternative method to compare two assembly functions. We flatten the JSON structure into a set, where each element is the full path concatenated with the value of each field in the object. Jaccard similarity (Intersection over union) is used to obtain a similarity score when comparing two such sets. That is, the number of identical key-value pairs in the JSON structure is divided by the total number of key-value pairs, giving a value between 0 and 1 which we use as the similarity score.

3.4 Dataset

The dataset used is made of a varied set of executables, so that it represents the diversity found in real world software. Our method is not trained, and so our dataset is only needed for evaluations. It is composed of seven open source binaries: BusyBox [44], GNU coreutils [10], curl [38], ImageMagick [8], OpenSSL [7], PuTTY [40], and SQLite [20]. All have permissive licenses that allow their use in our evaluations.

All binaries were compiled using gcc for the following architectures: x86-64, arm, mips, powerpc. For each architecture, executables were generated for all optimization levels (00 to 03), stripped of debug symbols. The compiled binaries were disassembled using IDA Pro and separated into individual functions, yielding 383,658 assembly functions. Functions consisting of less than three instructions were not included as part of the dataset, because of their triviality.

Pairs of equivalent functions from the same platform but distinct optimization levels were made for cross optimization evaluation, and pairs from the same optimization level but different platforms were made for cross platform evaluation. For example, the MD5Init functions in Figure 1 compiled with -O0 (left) and -O3 (right) for x86-64 form a pair for cross optimization retrieval, whereas functions compiled with -O0 for x86-64 (left) and mips (center) form a pair for cross architecture retrieval.

4 Experiments

Our experiments are run on a virtual machine with 8 Intel Xeon Gold 5218 cpu cores, 100 GB of RAM, and two NVIDIA Quadro RTX 6000 GPUs each having 24 GB of RAM.

The following experiments are performed: First, we select the LLM best suited for our subsequent evaluations. Second, we compare our method against other state-of-the-art ML based approaches for BCSO on our dataset. Third, we perform 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 any generic embedding model yields significantly better results than 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 [11, 18, 33, 45, 46]. 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 the same optimization level. The feature set generated for function a_1 is compared for similarity with the features of each 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 with Equation 1.

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

The Recall@1 metric is calculated by dividing the number of successful recalls by the total number of pairs, and the MRR metric is calculated by averaging the reciprocal rank of each pair.

4.2 LLM Selection

Before evaluating our method against the baselines, we compare different LLMs available to select our backbones for the remaining experiments. For the local model, three options are considered: Qwen2.5 Coder [21] with sizes 0.5B, 1.5B, 3B, and 7B; Gemma 3 [42] with sizes 1B and 4B; and Qwen3 4B [43]. We preselected these models because they are open

Model	00-03	02-03	arm-x86-64
Qwen2.5 Coder 7B	0.558	0.725	0.498
Qwen3 4B	0.550	0.850	0.564
Gemma 3 4B	0.571	0.839	0.594

Table 2: MRR results of the selected evaluations for local model selection, using a pool size of 100.

Model	00-03	02-03	arm-x86-64
Gemini 2.5 Flash	0.674	0.865	0.766
GPT 4.1 Mini	0.662	0.811	0.755

Table 3: MRR results of the selected evaluations for commercial model selection, using a pool size of 100.

source, small enough to fit in most modern GPUs when quantized, and small enough to fit within our GPU with 24 GB of VRAM unquantized. We did not consider mixture-of-experts models because they are much larger, and the specificity of our workload is likely to result in only a subset of expert being heavily used. We selected a small set of tasks that are representative of the extensive experiments conducted against the baselines. There are two cross optimization evaluations, and one cross architecture evaluation.

In all experiments on local models, the input context size is limited to 4096 tokens, and the maximum output tokens to generate is set to 512. Very large assembly functions that do not fit within the input tokens are truncated. The more recent Qwen3 and Gemma 3 models perform better for their size than Qwen2.5 Coder. They match or surpass it in most metrics while being around 40% smaller. The three models perform inference in a similar timeframe. On one of our GPUs, Qwen2.5 7B takes 10 seconds on average to generate an analysis for one assembly function. For smaller models such as Qwen2.5 0.5B and 1.5B, inference is faster since multiple queries can be batched on the same GPU (because of lower memory consumption). In all of our experiments, we parallelize analysis generation across our four GPUs. In practice however, queries are usually performed for a single assembly function, so a single GPU is needed. We selected Qwen2.5 Coder for the remaining experiments because it provides many small sizes for our ablation study, and is more stable than the newer Gemma 3 and Qwen3 models.

For the commercial model, we preselected GPT 4.1 Mini [35] and Gemini 2.5 Flash [41]. These were chosen mainly because of their low cost and availability. The same subset of evaluations as for local models is performed to determine the model to use for the remaining evaluations.

To provide a similar environment to local models, functions are truncated to a maximum length of 128 instructions. We selected Gemini 2.5 Flash because it performs best and offers a better infrastructure. Compared to GPT 4.1 Mini’s 10 seconds latency per request, Gemini is almost able to handle a request

every second, making it easier to iterate on our evaluations.

4.3 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. We first present the baselines and then show our results.

Baselines. There are five baselines, presented in the same order as in Table 5.

Order Matters [50] combines a BERT language representation model [9] along with two control flow graph embedding models 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 semantics 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 architecture retrieval.

SAFE [33] first encodes each instruction of an assembly function into a vector, using the word2vec model [34]. Using a Self-Attentive Neural Network [29], SAFE then converts the sequence of instruction vectors from the assembly function into a single vector embedding for the function. Like all other baselines, SAFE requires pre-training, and can perform cross architecture similarity detection. However, it was only trained on the `AMD64` and `arm` architectures.

Asm2Vec [11] inspired the SAFE model. It 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 architecture [27] (an extension of word2vec that embeds entire documents) to generate an embedding for the assembly function.

A more recent BCSD model, PalmTree [28], also bases its work on the BERT model [9]. 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) It is also trained on context window prediction, that is, predicting whether two instructions are found in the same context window of an assembly function. (3) Finally, the model is trained on def-use prediction—predicting whether there is a definition-usage relation between two instructions. This method supports both cross architecture and cross optimization tasks, although its reference implementation is only trained on cross compiler similarity detection.

The model CLAP [45] uses the RoBERTa [30] base model to perform assembly function embedding. It is adapted for assembly instruction tokenization, and directly generates an embedding for a whole assembly function. It also comes with a text embedding model, so that classification can be performed using human readable classes. The class to match for is embedded with the text model, and the assembly function with the assembly embedding model. The generated embeddings can be compared with cosine similarity to calculate whether the assembly function is closely related or opposed to the embedded category. This model was only trained for the `x86-64` architecture compiled with `gcc`, although it supports other architectures and could be trained on them.

Results. We present the results of the baselines and our method evaluated on both the largest local model, Qwen2.5-Coder 7B, and the commercially deployed model, Gemini 2.5 Flash. As evident in Table 4, the hardest retrieval task is between optimization levels 0 and 3, highlighting the substantial difference between unoptimized and maximally optimized code (see Figure 1). At optimization level 0, functions perform a lot of unnecessary actions, such as extensively moving data between registers and perform conditional evaluation of expressions that return a constant value. The generated code 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 for the targeted hardware, while keeping the observable behavior of the program untouched.

For the most part, the baselines perform worse than expected on the evaluations. As our own dataset is used rather than the one each baseline is pre-trained on, overfitting in the training process may be the cause of this poor performance, as also observed by Marcelli et al. [32]. Compared to the baselines, one of our method’s advantage is that it requires no specific training other than the pre-training process performed by the model developers. It readily achieves good results, meaning our method generalizes well to unseen settings. The stability of our method is also remarkable. It consistently performs well, compared to some of the baselines that perform excellently on some tasks, but poorly on others.

4.4 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 set or embedding. Hence, methods that

Model	MRR						Recall @ 1					
	00-01	00-02	00-03	01-03	02-03	average	00-01	00-02	00-03	01-03	02-03	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	0.059	0.063	0.057	0.068	0.051	0.060
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 4: 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
Order Matters	0.007	0.007	0.007	0.007	0.002	0.000	0.001	0.001
SAFE	0.239	0.187	0.196	0.207	0.081	0.059	0.064	0.068
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
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 5: 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` for the architectures specified in each column. Three examples are provided with our prompt.

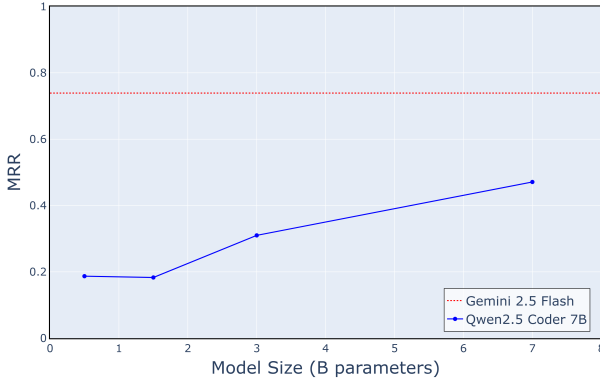


Figure 5: MRR performance for cross optimization retrieval against the number of parameters in the LLM. All assembly functions are compiled with `gcc` for the `arm` architecture. Retrieval is performed between optimization levels 0 and 1 with a pool of 1000 assembly functions. Three examples are provided with our prompt.

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 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 new 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 benefits from 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. Also, 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 slightly surpasses the baselines, but more work in this area is clearly still required. The recall@1 metrics show that the our method is able to rank the the correct assembly fragment in first place only 42% of the time on average.

4.5 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 [25], in that increasing the model size does significantly improve the results generated.

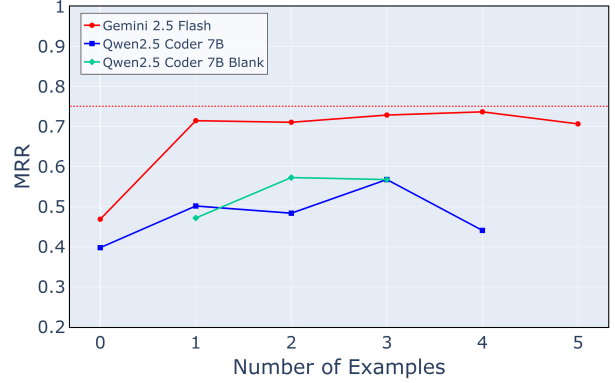


Figure 6: MRR performance for cross optimization retrieval against the number of examples provided in the prompt. Functions are compiled using `gcc` for the `x86-64` architecture. A pool of 100 assembly functions is used, with retrieval between optimization levels 0 and 3 for Gemini 2.5 Flash, and 0 and 1 for Qwen 2.5 7B.

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 the local models, and may use a mixture-of-experts architecture, based on previous Gemini model architectures.

4.6 Ablation on examples

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 shot prompting [5]. 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 three. 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 two or three examples are provided.

Another interesting result is the fact that when providing enough examples, the system prompt has very little impact

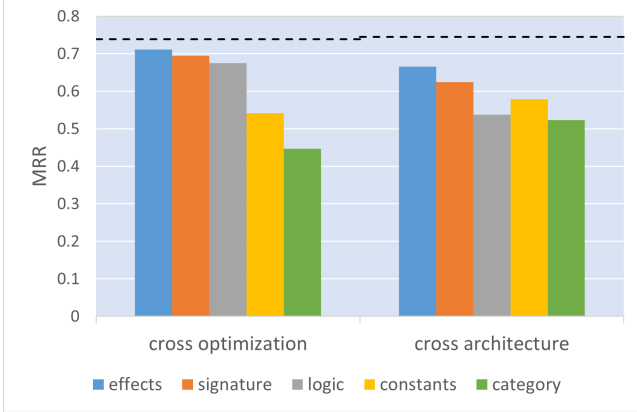


Figure 7: MRR performance with one section removed from the prompt, using Gemini 2.5 Flash. Cross optimization retrieval is done between fragments at optimization levels 0 and 3, with all functions compiled for the x86-64 architecture. Cross architecture retrieval is done between functions compiled for arm and x86-64, all 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 smaller bar means the prompt section has more impact on the results.

on the retrieval scores. This is depicted by the green data points in Figure 6, where only examples are provided with an empty string as the system prompt. This suggests that the examples themselves contain enough information for the model to perform the analysis, without being explained exactly how.

With our experimental configuration, providing four 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 language model. As such, most assembly functions are too large to fit in the remaining tokens and are thus truncated, which loses information about our query.

The dotted line in Figure 6 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 using the built-in capabilities of the API. The scores being very similar 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.

4.7 Ablation on the prompt used

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 0.30 on the MRR metric, when evaluated on the the hardest cross optimization task. The notable constants sections brings an increase of almost 0.20 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. As seen on Figure 7, 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.

4.8 Combined Method

Our method has shown to be an excellent generalist on BCSD retrieval tasks. Nevertheless, models trained for assembly function embedding remain beneficial in specific subdomains, such as supporting niche architectures or understanding similarity through specific obfuscation methods. State-of-the-art embedding models are much smaller than our method in terms of parameter count. For instance, the CLAP [45] model baseline is only 110M parameters, compared to the billions required to perform our method. Their small size potentially allows for CPU execution on small workloads. The cost of training such models is still very expensive, but once trained, they are much faster than LLM inference.

To achieve the best of both worlds, we experiment with combining the similarity scores from an embedding model and our prompting method. Our evaluation 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 similarity s_a is calculated using jaccard similarity. Both similarity scores are then combined with equal weight.

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

In a real life scenario, combining both methods could be performed by using our textual representation as a pre-filter, and then calculating the embedding representation. As such, the scalability and maintainability of the underlying BCSD

database is maintained, while allowing a flexible choice of embedding based BCSD model. That is, an inverted index database is maintained for the pre-filter query. Once this query is completed, the top- k results can be refined using an embedding model, by calculating the embeddings only for those k candidates. With this approach, the decision to use an embedding model, and which one to use, can be made dynamically.

To provide a method that keeps all of the advantages of our presented work, we use Qwen3-Embedding 4B [43] as the embedding model for this experiment. As such, the combination still does not require any training nor fine-tuning. As shown in Table 6, 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, since the training phase is performed by the open source model developers.

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

5 Related Work

5.1 Early BCSD methods

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. [36] Control flow graph analysis and comparison [12, 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 [2, 24, 37] are less time consuming, but tend to fixate on the syntactic elements and their ordering rather than the semantics.

Dynamic analysis. This method family consists of analyzing the features of a binary or code fragment by monitoring its runtime behavior. 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. [31] 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 [13].

5.2 Other code matching approaches

Code Representations. Our research focuses on methods of binary similarity that use the assembly code (and possibly the control flow graph) retrieved from the binary to perform similarity detection. However, other approaches exist that use

an alternate representation language, so that binaries compiled for all architectures are extracted into the same representation language. Some methods turn to the well known LLVM-IR [?, ?, ?] as representation language, and others prefer the VEX intermediate representation [?, ?].

Full Program Similarity. An alternate approach to binary matching is to compare whole binaries, instead of their individual functions. This type of method constructs a fingerprint for the whole binary as compared to having one embedding per function. This approach is primarily used for malware identification [?, ?, ?, ?], but also for identifying known vulnerabilities [?, ?].

5.3 LLM based software analysis

Researchers have found many related applications of language models to binary and software analysis. We give a sparse overview of the progress that has been made in this area of research. For vulnerability detection, FuncVul [17] is a GraphCodeBERT model fine-tuned to detect whether a provided source function is vulnerable. The dataset generation process makes use of a LLM to rewrite each function using generic function and variable names. The LLM is also used in their method to predict the lines of code on which the vulnerability is found. LLM4Decompile [39] tries to utilize LLMs for decompilation. An open source LLM is fine-tuned to learn the source code representation of an assembly function, and the model is then evaluated on its decompilation capabilities. Fang et al. [14] measure the competency of LLMs for source code analysis, with a focus on obfuscated source code. Their results match our findings, that LLMs are efficient at code analysis, but they also demonstrate that language models still struggle when faced with cases of advanced obfuscation techniques. LLM based fuzzing is another area that has recently gained interest. For instance, Asmita et al. [4] use a LLM to generate the initial seed of a fuzzing pipeline on the BusyBox [44] executable.

6 Conclusion

Our method for BCSD is 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,

	Gemini 2.5 Flash	Qwen Embedding	Combined
00 - 01	0.646	0.640	0.910
00 - 02	0.579	0.554	0.843
00 - 03	0.485	0.518	0.759
01 - 03	0.618	0.640	0.855
02 - 03	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 6: 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.

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 confident that our method brings significant improvements to the current landscape of BCSD, by resolving 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, and remains accurate compared to approximated nearest neighbor search methods.

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 opportunities are outlined here.

6.1.1 Fine-tuning

As stated, our method does not require fine-tuning, and was not evaluated on fine-tuned performance. 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 [6]. With distillation, a large model is used to train or fine-tune a much smaller model. For example in our experiments, Gemini 2.5 Flash could have been used to fine tune the small Qwen2.5-Coder models. Since there is no ground truth in text generation, the distillation process uses the large model’s

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 Qwen2.5 7B at assembly understanding. We believe the distillation process could considerably reduce this gap.

6.1.2 Output Format

Our method uses the JSON output format so that the output generated is easily comparable and a similarity metrics can be deduced. This format was chosen because it is ubiquitous and its syntax is straightforward to produce for LLMs, and easily understandable for humans. However, finding the best trade-off between output interpretability, compute requirements, and similarity detection capabilities remains an open problem. One possibility is to loosen the restrictions on the output format and use a sentence similarity method to determine the similarity between function analyses. On the contrary, another possibility is to use the analysis capabilities at a smaller scale, for example analyzing single basic blocks and combining these analyses into a more comprehensive structure for a whole assembly function. In any case, more experimentation is required to find optimal balance.

6.1.3 Reasoning models

A recent class of LLMs are fine tuned to natively use chain of thought inference [47] 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 [26, 47, 48] on hard problems. It may be worth evaluating whether this style of inference performs better than non reasoning models (as used in our research) on assembly function analysis.

6.1.4 Other integrations

Our work proves the competency of LLMs in understanding and analyzing assembly code. From this point, different approaches to incorporating LLMs into assembly analysis workflows can be considered. First, instead of using LLMs for feature extraction, they could be used to explain the similarity between two functions, once this similarity has been determined using an unexplainable method such as embeddings. Furthermore, it could be used as a decision maker as to whether the two code fragments are similar enough to be considered clones. Often times, the assembly function used as query is not even part of the BCSD database. The remarkable versatility of LLMs underscores the need for further research to determine where they can effectively support the reverse engineering workflow and where their utility remains limited.

Acknowledgments

The USENIX latex style is old and very tired, which is why there's no \acks command for you to use when acknowledging. Sorry.

Do not include any acknowledgements in your submission which may deanonymize you (e.g., because of specific affiliations or grants you acknowledge)

Ethical Considerations

An analysis of our research is performed under the four ethical principles suggested.

Respect for persons. Our research is not conducted on human subjects, and thus respect for persons is not a concern.

Beneficence. The objective of our work is to ease reverse engineering to enhance and ensure the security of software. We use open source components wherever possible, such as our dataset and the local language models. This approach enables us to publicly release all artifacts used in our research, which allows for reproducibility and new research to directly build on our results.

Justice. To the best of our knowledge, our research did not involve the unfair treatment of entities, groups, or individuals and thus justice is not a concern.

Respect for Law and Public Interest. All research steps performed are legal. All licenses of the software used in our research were respected. We believe it is in the public’s best interest to freely have access to our research, but we recognize that our research may be used by attackers to ease their reverse engineering efforts and discover vulnerabilities that could be exploited on the public. However, it is widely understood that developing software as closed-source is not a valid security measure. Software providers must not rely on the complexity of reverse engineering to secure their software. Furthermore, it is beyond our control to ensure that others adhere to fundamental software security guidelines.

Open Science

Consistent with the USENIX Security open science policy, the repository used for experimentation of our method is publicly available [TODO]. This constitutes the full implementation of our methods that can be reproduced on any open source model, and the implementation for commercial LLM providers. The dataset used throughout this research is also available at that same repository [TODO].

References

- [1] National Security Agency. Ghidra software reverse engineering framework, 2025-07-31. URL: <https://github.com/NationalSecurityAgency/ghidra>.
- [2] Farouq Al-Omari, Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. Detecting clones across microsoft .net programming languages. In *2012 19th Working Conference on Reverse Engineering*, pages 405–414, 2012. doi:10.1109/WCRE.2012.50.
- [3] Alexandr Andoni, Piotr Indyk, and Ilya Razenshteyn. Approximate nearest neighbor search in high dimensions, 2018. URL: <https://arxiv.org/abs/1806.09823>, arXiv:1806.09823.
- [4] Asmita, Yaroslav Oliynyk, Michael Scott, Ryan Tsang, Chongzhou Fang, and Houman Homayoun. Fuzzing BusyBox: Leveraging LLM and crash reuse for embedded bug unearthing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 883–900, Philadelphia, PA, August 2024. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity24/presentation/asmita>.
- [5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL: <https://arxiv.org/abs/2005.14165>, arXiv:2005.14165.
- [6] Cristian Buciluundefined, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’06*, page 535–541, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1150402.1150464.
- [7] Mark Cox, Ralf Engelschall, Stephen Henson, Ben Laurie, and Paul Sutton. Openssl, 2025-08-05. URL: <https://www.openssl.org/>.
- [8] John Cristy. Mastering digital image alchemy, 2025-07-13. URL: <https://imagemagick.org/>.
- [9] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. *SIGPLAN Not.*, 53(2):392–404, March 2018. doi:10.1145/3296957.3177157.
- [10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL: <https://arxiv.org/abs/1810.04805>, arXiv:1810.04805.
- [11] Alex Deymo, Jim Meyering, Paul Eggert, Padraig Brady, Bernhard Voelker, and Collin Funk. Coreutils - gnu core utilities, 2025-04-09. URL: <https://www.gnu.org/software/coreutils/>.

- [12] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe 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)*, pages 472–489, 2019. doi:10.1109/SP.2019.00003.
- [13] Thomas Dullien. Graph-based comparison of executable objects. 2005. URL: <https://api.semanticscholar.org/CorpusID:2001486>.
- [14] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, San Diego, CA, August 2014. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele>.
- [15] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. Large language models for code analysis: Do LLMs really do their job? In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 829–846, Philadelphia, PA, August 2024. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity24/presentation/fang>.
- [16] Mohammad Reza Farhadi, Benjamin C.M. Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 78–87, 2014. doi:10.1109/SERE.2014.21.
- [17] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS ’17, page 346–359, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3052973.3052995.
- [18] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’16, pages 480–491, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978370.
- [19] Yi Gui, Yao Wan, Hongyu Zhang, Huifang Huang, Yulei Sui, Guandong Xu, Zhiyuan Shao, and Hai Jin. Cross-language binary-source code matching with intermediate representations, 2022. URL: <https://arxiv.org/abs/2201.07420>, arXiv:2201.07420.
- [20] Sajal Halder, Muhammad Ejaz Ahmed, and Seyit Camtepe. Funcvul: An effective function level vulnerability detection model using llm and code chunk, 2025. URL: <https://arxiv.org/abs/2506.19453>, arXiv:2506.19453.
- [21] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. Code is not natural language: Unlock the power of Semantics-Oriented graph representation for binary code similarity detection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1759–1776, Philadelphia, PA, August 2024. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity24/presentation/he-haojie>.
- [22] hex rays. Ida pro, 2025-02-28. URL: <https://hex-rays.com/ida-pro>.
- [23] D. Richard Hipp. Sqlite, 2025-07-30. URL: <https://sqlite.org/>.
- [24] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL: <https://arxiv.org/abs/2409.12186>, arXiv:2409.12186.
- [25] Piotr Indyk and Haike Xu. Worst-case performance of popular approximate nearest neighbor search implementations: Guarantees and limitations, 2023. URL: <https://arxiv.org/abs/2310.19126>, arXiv:2310.19126.
- [26] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.
- [27] Steven Jilcott. Scalable malware forensics using phylogenetic analysis. In *2015 IEEE International Symposium on Technologies for Homeland Security (HST)*, pages 1–6, 2015. doi:10.1109/THS.2015.7225311.
- [28] Wesley Jin, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. Binary function clustering using semantic hashes. In *2012 11th International Conference on Machine Learning and Applications*, volume 1, pages 386–391, 2012. doi:10.1109/ICMLA.2012.70.

- [29] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL: <https://arxiv.org/abs/2001.08361>, arXiv:2001.08361.
- [30] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023. URL: <https://arxiv.org/abs/2205.11916>, arXiv:2205.11916.
- [31] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents, 2014. URL: <https://arxiv.org/abs/1405.4053>, arXiv:1405.4053.
- [32] Qiang Li, Dawei Tan, Xin Ge, Haining Wang, Zhi Li, and Jiqiang Liu. Understanding security risks of embedded devices through fine-grained firmware fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 19(6):4099–4112, 2022. doi:10.1109/TDSC.2021.3119970.
- [33] Xuezixiang Li, Yu Qu, and Heng Yin. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21*, page 3236–3251, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460120.3484587.
- [34] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding, 2017. URL: <https://arxiv.org/abs/1703.03130>, arXiv:1703.03130.
- [35] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019. URL: <https://arxiv.org/abs/1907.11692>, arXiv:1907.11692.
- [36] Zian Liu. Binary code similarity detection. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1056–1060, 2021. doi:10.1109/ASE51524.2021.9678518.
- [37] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116, Boston, MA, August 2022. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli>.
- [38] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Safe: Self-attentive function embeddings for binary similarity, 2019. URL: <https://arxiv.org/abs/1811.05296>, arXiv:1811.05296.
- [39] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality, 2013. URL: <https://arxiv.org/abs/1310.4546>, arXiv:1310.4546.
- [40] OpenAI. Gpt-4 technical report, 2024. URL: <https://arxiv.org/abs/2303.08774>, arXiv:2303.08774.
- [41] Panpan Qi, Zhaoqi Zhang, Wei Wang, and Chang Yao. Malware detection by exploiting deep learning over binary programs. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 9068–9075, 2021. doi:10.1109/ICPR48806.2021.9412227.
- [42] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. Malware detection by eating a whole EXE. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, volume WS-18 of *AAAI Technical Report*, pages 268–276. AAAI Press, 2018. URL: <https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16422>.
- [43] Liting Ruan, Qizhen Xu, Shunzhi Zhu, Xujing Huang, and Xinyang Lin. A survey of binary code similarity detection techniques. *Electronics*, 13(9), 2024. URL: <https://www.mdpi.com/2079-9292/13/9/1715>, doi:10.3390/electronics13091715.
- [44] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K. Peña, Borja Sanz, Carlos Laorden, and Pablo G. Bringas. Idea: opcode-sequence-based malware detection. In *Proceedings of the Second International Conference on Engineering Secure Software and Systems, ES-SoS'10*, page 35–43, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-11747-3_3.
- [45] Noam Shalev and Nimrod Partush. Binary similarity detection using machine learning. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS '18*, page 42–47, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3264820.3264821.
- [46] Daniel Stenberg. curl - command line tool and library for transferring data with urls, 2025-07-16. URL: <https://curl.se/>.

- [47] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. LLM4Decompile: Decompiling binary code with large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3473–3487, Miami, Florida, USA, November 2024. Association for Computational Linguistics. URL: <https://aclanthology.org/2024.emnlp-main.203/>, doi:10.18653/v1/2024.emnlp-main.203.
- [48] Simon Tatham. Putty, 2025-02-08. URL: <https://www.chiark.greenend.org.uk/~sgtatham/putty/>.
- [49] Google Team. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025. URL: <https://arxiv.org/abs/2507.06261>, arXiv:2507.06261.
- [50] Google Team. Gemma 3 technical report, 2025. URL: <https://arxiv.org/abs/2503.19786>, arXiv:2503.19786.
- [51] Qwen Team. Qwen3 technical report, 2025. URL: <https://arxiv.org/abs/2505.09388>, arXiv:2505.09388.
- [52] S. VenkataKeerthy, Soumya Banerjee, Sayan Dey, Yashas Andaluri, Raghul PS, Subrahmanyam Kalyanasundaram, Fernando Magno Quintão Pereira, and Ramakrishna Upadrasta. Vexir2vec: An architecture-neutral embedding framework for binary similarity. *ACM Trans. Softw. Eng. Methodol.*, March 2025. Just Accepted. doi:10.1145/3721481.
- [53] Denys Vlasenko, Bernhard Reutner-Fischer, and Rob Landley. Busybox, 2024-09-27. URL: <https://busybox.net/>.
- [54] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi 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, ISSTA 2024*, page 503–515, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3650212.3652145.
- [55] Jialai Wang, Chao Zhang, Longfei Chen, Yi Rong, Yuxiao Wu, Hao Wang, Wende Tan, Qi Li, and Zongpeng Li. Improving ML-based binary function similarity detection by assessing and deprioritizing control flow graph features. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4265–4282, Philadelphia, PA, August 2024. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity24/presentation/wang-jialai>.
- [56] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL: <https://arxiv.org/abs/2201.11903>, arXiv:2201.11903.
- [57] Cangshuai Wu, Jiangyong Shi, Yuexiang Yang, and Wenhua Li. Enhancing machine learning based malware detection model by reinforcement learning. In *Proceedings of the 8th International Conference on Communication and Network Security, ICCNS '18*, page 74–78, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3290480.3290494.
- [58] Tianhao Wu, Janice Lan, Weizhe Yuan, Jiantao Jiao, Jason Weston, and Sainbayar Sukhbaatar. Thinking llms: General instruction following with thought generation, 2024. URL: <https://arxiv.org/abs/2410.10630>, arXiv:2410.10630.
- [59] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: Security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472, 2017. doi:10.1109/ICSE.2017.49.
- [60] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(01):1145–1152, Apr. 2020. URL: <https://ojs.aaai.org/index.php/AAAI/article/view/5466>, doi:10.1609/aaai.v34i01.5466.
- [61] Zhongtang Zhang, Shengli Liu, Qichao Yang, and Shichen Guo. Semantic understanding of source and binary code based on natural language processing. In *2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, volume 4, pages 2010–2016, 2021. doi:10.1109/IMCEC51613.2021.9482032.