Practical Source Code Recovery from Binary Functions Using Anchor-Based Retrieval and LLM Reasoning

Charles Edward Gagnon[1][0009−0008−4647−4398], Steven H. H. Ding[1][0000−0003−4513−200X], Philippe Charland[2][0000−0003−4051−9942], and Benjamin C. M. Fung[1][0000−0001−8423−2906]

[1] McGill University, Montreal, QC H3A 0G4, Canada
charles.e.gagnon@mail.mcgill.ca, {steven.h.ding, ben.fung}@mcgill.ca
[2] Defence Research and Development Canada

Abstract. We present a practical pipeline for recovering source code from stripped binary functions by combining reverse engineering, anchor-based source code retrieval, and large language model reasoning. Our binary-to-source-code retrieval method attempts to identify the source function from a source code database, rather than generating approximate decompiled pseudocode. It extracts anchors such as strings, constants, external calls, and available function names using Ghidra, retrieves candidate files via an inverted-index search database, narrows candidates to likely function snippets, and re-ranks them with a large language model (LLM) based on disassembly, decompiled code, and source metadata. Confident matches can also serve as anchors in later passes. In an evaluation backed by our high-fidelity source code database on a stripped, optimized `tcpdump` binary, our proposed binary-to-source matching method achieves 95.2% assembly instruction coverage. Experiments on a GitHub-based retrieval database showed lower performance with 35.5% instruction coverage on average, mainly due to retrieval misses. These results show that source-level binary recovery excels with high-quality databases and remains a useful tool in noisy environments.

Keywords: Binary-to-source code matching · Reverse engineering · Source code recovery · Binary analysis · Anchor-based retrieval · Large language models.

1 Introduction

With the ever-increasing rate of software production, reverse engineering unknown executables has become a major bottleneck for cybersecurity. Organizations must analyze a growing number of binaries originating from commercial software, malware samples, firmware images, and third-party dependencies. While software development has benefited greatly from automation and advancements in tooling, the reverse engineering process remains largely dependent on

expert analysts and manual investigation. As a result, the time required to understand and assess compiled software increasingly limits the speed at which security teams can investigate incidents and detect threats.

Binary code reverse engineering is a challenging task that, even today, requires heavy human intervention. No automatic reverse engineering tool exists because compilers are fundamentally irreversible functions [1]. A compiler strictly maintains the semantic rules of the underlying source code, but removes all other helpful information. With optimizations enabled, a compiler will liberally get rid of object identifiers, control-flow structures, logical ordering, data structures, and even whole functions. As such, even a perfect system cannot recover as much information from the compiled binary as what was initially present in the source code. There exists a variety of tools that try to automatically obtain pseudocode from binary code. These are usually categorized as decompilers [4]. However, more than a decompiler is needed to build a deep understanding of unknown software. The decompiled output represents the semantic meaning of the software, but lacks the structure, comments, and identifiers used by the developer to make sense of the code.

In this work, we present a method that fully recovers the source code of unknown binary functions. Our approach takes advantage of the significant presence of open source software in modern applications [16,15]. Our method is practical because it readily supports massive open databases such as the GitHub API and does not require the maintenance of custom machine learning models.

2 Related Works

Binary to source code matching. Numerous methods that match binary to source code already exist. A recurring theme is the use of disassembler such as Ghidra [17] or IDA Pro [10] to extract binary function features. One of the first contributions to this field is RESource [19]. This method uses the extracted features to perform web queries. The query responses are parsed and injected as code comments into the disassembled binary source. BinPro [14] brought significant improvements by introducing machine learning in the feature extraction and matching steps. The weights associated with each feature are learned rather than hard-coded, and the queries are performed in dedicated source code databases rather than on the internet. B2SFinder [26] improved generalizability by extracting a wider range of features, rather than only focusing on string constants. These include numeric constants, exported symbols, function names, enumeration types and conditional branching. Many other methods branded under library usage detection [23,7] or software bill of materials (SBOM) generation [24,18,3] perform a very similar task. Instead of matching a specific source file to a binary routine, these methods only determine whether a known library is used by the binary code. The granularity of these methods is a common limitation. In binary to source code matching, most methods only report whether a library or source file was used, without pointing to the exact source code snippet.

Furthermore, most algorithms cannot provide a rationale behind the matching outputs.

Binary to binary matching. A related area is binary code to binary code matching, also known as binary code similarity detection. This area of research attempts to find the closest matching binary function in a database, rather than filtering through a source code database. Current state-of-the-art methods make heavy use of machine learning and natural language processing to achieve high accuracy [22,13,12,5,25]. Similar to our proposed method, some directly use LLMs to perform feature extraction [21], feature analysis [8], or candidate selection [6]. Other methods use the source code as part of model training, and can determine the similarity between the binary source and a descriptive label of the code [22]. These approaches are limited by the expressiveness of the matched binary code. If the source code corresponding to the matched binary function is unknown, then identifying a similar binary function does not substantially advance the reverse engineering task. Another limitation is the capability of the model to match code compiled for different architectures and with varying optimization levels.

Decompilation. Decompilation tools also perform binary code to source code matching, but do so without the help of a source code database. These methods perform static analysis of disassembled binary code and generate pseudocode as output. Recent research shows that LLMs can significantly enhance traditional template-based decompilation pipelines. Models trained specifically on parallel binary-to-source corpora [20] leverage natural language capabilities to predict high-level variable names and generate more readable pseudocode than heuristic-based decompilers. A recurring limitation is that LLM-based decompilers are non-deterministic, which can make the output unreliable and hard to verify.

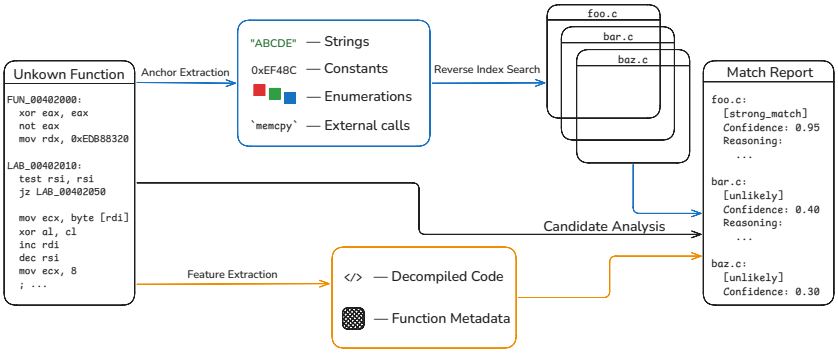

Fig. 1. Overview of our method. An unknown function is first disassembled and its features are extracted. The anchors are used to perform the initial database query. The disassembled code, along with information-dense features, is used to re-order the top-k candidates. In our experiments, k is set to 12.

3 Methodology

We formulate binary-to-source recovery as a retrieval and verification problem. Given a stripped binary function b, the goal is to identify the source-level function s from a source code database D that most likely produced b after compilation. Unlike decompilation, which attempts to synthesize readable pseudocode from binary instructions alone, our method searches for an existing source implementation and verifies whether it corresponds to the observed binary routine. Our method assumes that the target source code, or a sufficiently similar implementation, is present in the backing database. It does not expect the presence of debug symbols, although it can use them if available.

Our pipeline is shown in Figure 1. First, static analysis is applied to the target binary function to extract compact retrieval features and denser function features. Second, retrieval features are used as anchors to retrieve candidate source files from a source code database. Third, the retrieved files are narrowed to function snippets and ranked based on the number of anchors matched. Finally, a large language model verifies the most promising candidates and re-ranks them by comparing the dense function features with each candidate source function. Confident matches can then be propagated through the binary call graph as additional anchors for later passes.

3.1 Feature Extraction

The first stage extracts features from the target binary function using Ghidra [17]. We distinguish between two classes of features: *anchors*, which are used for retrieval, and dense features, which are reserved for later verification and re-ranking.

Anchors are features used in the initial source code database query to find relevant source files. They include strings and numerical constants, external calls, and function names. The most reliable anchor is a string constant. It provides heavy filtering capabilities and a very low false positive rate. Numerical constants are also useful, but their smaller domain makes them more susceptible to false positives. They are especially useful in functions that use unique values, such as hashing routines and other cryptographic primitives. External function calls are less unique but remain useful when combined with others. These are usually `libc` function calls or compiler builtins, such as `memset` or `memcpy`. Lastly, internal function names are highly effective when available, although they are usually absent from stripped production binaries.

Dense features are not used directly in the initial database query. Instead, they are used during candidate verification. These include the disassembled instructions, the decompiled pseudocode, and metadata recovered by Ghidra. These features provide a richer description of the binary function, but they are too verbose to be used directly as inverted-index search keys.

3.2 Source Code Retrieval

The second stage uses the extracted anchors to retrieve candidate source files from the backing database. To remain scalable, the database is accessed through an inverted-index search interface. The query consists of the anchors extracted from the binary function and the database returns source files ranked according to their textual relevance to those anchors. This abstraction allows the method to operate over different source code databases. In a controlled enterprise setting, the database can be implemented as a domain-specific and curated index of source files. In a large-scale real-world setting, the same interface can be implemented using external code search services such as the GitHub API. The retrieval stage does not depend on a specific database implementation, but its effectiveness depends on the completeness, ranking quality, and noise level of the backing corpus.

3.3 Candidate localization

Once matching source files are found, the code corpus is reduced to only contain relevant snippets. Our method uses a large combination of techniques to handle a number of issues. The first issue comes with constants and preprocessor macros. Standard code style guidelines [2] recommend constants to be declared before their use, usually at the very beginning of a file. This poses an issue because naive snippet extraction would very often return inconsequential portions of the source file. To alleviate this issue, a preprocessor analyzes the statement containing the anchor and decides whether this statement constitutes a definition or a use site. If the anchor is found at a definition site, all use sites of the identifier to which the constant is assigned are added to the candidates, and the definition site is ignored. Another issue stemming from compiler optimizations is function inlining. Anchors can point to a function that is not actually part of the compiled binary because it was inlined in the body of the calling function. As such, our method uses heuristics to determine whether a function is likely to be inlined by the compiler (function size, presence of inline tags, and control flow complexity). If the target function is deemed likely to be inlined, the function name is added to the set of anchors. This largely increases the effectiveness of our method. It helps verification to find the full source code function linked to the compiled binary routine, rather than a smaller function that was inlined. With these techniques, our method is able to reliably find source snippets that relate to the anchors. Snippets use function boundary detection to fit whole functions as the result. The number of anchors in the snippet and the ratio of anchors to lines of source are used to rank source snippets in order of importance.

3.4 Function similarity reasoning

The novelty of our method comes from the way in which it handles source snippet candidates. As highlighted in Figure 2, instead of relying on a simple

Decompiled code

```
undefined8 FUN_0010bd40(undefined8
↪  *param_1,long param_2)
{
char *__file;
undefined8 uVar1;
int iVar2;
uint uVar3;
FILE *pFVar4;
int *piVar5;
char *pcVar6;
size_t __n;
uint *puVar7;
int iVar8;

if ((*(int )(param_2 + 0x500) == 2) ||
(((int )(param_2 + 0x500) == 0 && ((char
↪  *)((long)param_1 + 9) == '\0')))) {
pFVar4 = fopen(__file,"wb");
if (pFVar4 != (FILE *)0x0) {
LAB_0010be15:
param_1[2] = pFVar4;
(undefined1 () [16])(param_1 + 3) =
↪  (undefined1  [16])0x0;
*(undefined2 )((long)param_1 + 10) = 0x101;
return 1;
}
}
else {
/* ... */
}

/* ... */
piVar5 = __errno_location();
pcVar6 = strerror(*piVar5);
FUN_00115b40(uVar1,"Failed to open the file
↪  %s: %s",__file,pcVar6);
return 0;
}
```

Source code

```
bool tool_create_output_file(
struct OutStruct *outs,
struct OperationConfig *config)
{
FILE *file = NULL;
const char *fname = outs->filename;
DEBUGASSERT(outs);
DEBUGASSERT(config);
DEBUGASSERT(fname && *fname);

if(config->file_clobber_mode ==
↪  CLOBBER_ALWAYS ||
(config->file_clobber_mode ==
↪  CLOBBER_DEFAULT &&
!outs->is_cd_filename)) {
/* open file for writing */
file = curlx_fopen(fname, "wb");
}
else {
int fd;
do {
/* ... */
}
}

if(!file) {
char errbuf[STRERROR_LEN];
warnf("Failed to open the file %s: %s",
↪  fname,
curlx_strerror(errno, errbuf,
↪  sizeof(errbuf)));
return FALSE;
}
outs->regular_file = TRUE;
outs->fopened = TRUE;
outs->stream = file;
outs->bytes = 0;
outs->init = 0;
return TRUE;
}
```

The candidate source in `curl/src/tool_cb_wrt.c` matches the decompiled function closely in control flow, error handling, and key string literals such as **“Remote filename has no length”**, **“out of memory”**, and **“Failed to open the file %s: %s”**. The source function `tool_create_output_file` implements the same logic of opening files with conditions on file existence, modes, and error reporting. The use of `open`, `fopen`, `fdopen`, `close`, `malloc`, `memcpy`, `curl_msnprintf`, and `errno` handling matches the decompiled code’s calls and flow. The numeric constants and offsets (e.g., `0x508`, `0x500`) correspond to field accesses in the source. The source function signature and parameter usage align well with the decompiled code’s `param_1` and `param_2` usage. This is a direct match to the decompiled function, not just a helper or macro.

Fig. 2. Example of successful retrieval from decompiled code to source code. The language model is able to precisely explain the reasoning behind the match.

ranking based on anchor density or other heuristics, our method uses the capabilities of LLMs to perform an in-depth analysis of each candidate. The representations extracted from the binary function are provided in the language model query. The response follows a specific schema, reporting the matching **score**, the **reasoning** explaining which elements of the function match and differ, and a final **verdict** that indicates the confidence of the model in the match.

A surprising benefit is that our method can also be applied iteratively. Once a binary function is confidently matched to a source function, the recovered source identity can serve as a new anchor for caller functions. For example, if a binary function at address x is matched to a source function f, then calls to x from other binary functions can be interpreted as calls to f. The name f can then be added to the anchor set of those caller functions. This secondary-pass mechanism allows information recovered from high-confidence matches to propagate through the call graph. It is especially useful for functions that contain few intrinsic anchors but call functions that have already been identified. This way, the method can recover additional source functions that would be difficult to retrieve from local features alone.

4 Experiments

To assess the capabilities of our method, we first experimented it in a controlled environment against our own database of open source projects. We then performed an experiment on the public GitHub API, showcasing how our method generalizes to any database already in production. The metrics presented for each experiment include the percentage of functions correctly identified in first place (Hit @ 1) and in the top three (Hit @ 3). We additionally provide the mean reciprocal rank (MRR)

$$\mathrm{MRR} = \frac{1}{|B|} \sum_{i=1}^{|B|} \frac{1}{\mathrm{rank}_i}$$

where B is the set of binary functions. Finally, we report the percentage of the binary that is fully reverse engineered by our tool (Coverage). That is, the percentage of assembly instructions that are part of a binary function that is correctly mapped to its source code function.

4.1 Controlled Environment Evaluation

For our first experiment, the `tcpdump` binary is reverse engineered with our approach against a private database containing source files from $8,061$ open source projects. The `tcpdump` binary is compiled with optimization level O2, for the `x86_64` architecture. A map between function addresses and their source is available as ground truth. To do so, the binary is first compiled with debug symbols. An address-to-function map that stores the original function name, the source file and line number is extracted from the binary using the `nm` tool [9]. Finally, the binary is stripped of all debug symbols and passed to our tool for

analysis. Results are reconstructed by comparing the output of our tool with the address-to-function map. If function names match and both functions are located in the same source file, then the function is considered correctly mapped. We selected GPT-5.5 for candidate verification in this experiment, seeking to obtain results that are not limited by language model capabilities.

Table 1. Evaluation metrics of our experiment on the `tcpdump` binary with a curated database.

Metric	**Value**
Queries Evaluated	815
Queries with Hit	714
Miss	101
Top 1	698
Top 3	714
Hit@1	0.8564
Hit@3	0.8761
MRR	0.8660
Coverage	95.2%

As evident from Table 1, our method performs remarkably well on classical reverse engineering of code compiled from `C++`. Our method alone covers 95% of all instructions in the binary with a correctly matched source function. Most failed matches stem from very small assembly routines that do not contain any reliable anchor for the initial database query to be effective. For instance, 101 functions from the `tcpdump` binary contained zero anchors, all of which could not return any meaningful results from the database. However, out of the 714 queries that did contain anchors, all valid matches were found in the top 3 candidates from the language model analysis. This is explained by the database containing very little noise (such as markdown documents, configuration files and logs). As such, queries are efficient, especially when unique anchors are provided. It is worth noting that with an unknown malware sample or proprietary binaries, our method is unable to reverse engineer functionality that is not previously found in the database. Yet, it is known that a vast majority of proprietary software constitutes code reuse of open source libraries [16,15]. Nevertheless, our method can still find close matches when the exact match is not found in the database, given a large enough dataset.

4.2 Real-World Use Case

Next, we evaluate the method in a real-world setting by replacing the curated database with the public GitHub code search interface. This experiment is intended to measure how the pipeline behaves when the backing database is large, noisy, and externally ranked. The set of binaries evaluated in this experiment consists of ubiquitous open source utilities: `BusyBox`, `curl`, `sqlite3`,

and `dropbear`. All binaries are compiled for Linux `x86_64` with optimization level O2. We also vastly reduce the size of the language model, going from GPT 5.5 (estimated 9.7T parameters [11]) to GPT 5 mini (estimated 410B parameters [11]).

Table 2. Evaluation metrics of our experiment using the GitHub API as the backing source code database.

	`BusyBox`	`sqlite3`	`curl`	`dropbear`	average
Hits @ 1	0.134	0.191	0.541	0.265	0.283
Hits @ 3	0.191	0.268	0.712	0.354	0.381
Instruction Coverage	18.7%	26.0%	67.1%	30.4%	35.5%
False Positives	0.068	0.140	0.329	0.159	0.174
Database Miss	0.751	0.668	0.130	0.577	0.531
Hits @ 1 among retrieved	0.536	0.577	0.622	0.625	0.590
Hits @ 1 no re-raking	0.080	0.068	0.336	0.245	0.182
Total	4182	1867	146	359	

The results in Table 2 are substantially weaker than those obtained in the controlled experiment. Across all four binaries, the average Hits @ 1 drops to 0.283 and the average instruction coverage drops to 35.5%. This reduction is expected. The GitHub API does not provide a clean, deduplicated corpus of source files, unlike our curated database. Instead, the retrieval stage must operate over a large and heterogeneous source code collection whose ranking is out of our control. As a result, many functions fail before the language model re-ranking stage is reached.

The dominant failure mode is database miss. On average, 53.1% of functions are classified as database misses, meaning that the correct source function was not present among the retrieved candidates. This is especially visible for `BusyBox`, `sqlite3`, and `dropbear`, where the database miss rates are 0.751, 0.668, and 0.577, respectively. These values indicate that the main bottleneck in the real-world setting is not only source-to-binary reasoning, but retrieval. If the correct file is absent from the candidate set, the re-ranker cannot recover the match. This interpretation is supported by the Hits @ 1 among retrieved metrics. When database misses are excluded, the average Hits @ 1 rises from 0.283 to 0.590. This shows that the real-world performance of the full system is tightly coupled to the stability and quality of the backing search engine. The comparison with the non-LLM baseline further clarifies the role of the language model. Without the language model re-ranker, the average Hits @ 1 is only 0.182. Adding the re-ranker increases this value to 0.283, and the improvement is especially large for `curl`, where Hits @ 1 rises from 0.336 to 0.541. Nevertheless, LLM reasoning cannot compensate for missing candidates.

5 Limitations and Conclusion

While our method performs excellently with a controlled data source and a performant model, the output is not as ideal in a noisy environment. First, having a very large data source such as the one provided by GitHub increases the potential of not finding the correct match in the first dozen results from the database. Furthermore, the GitHub database contains a lot of duplicates for popular projects because of repository forks, exacerbating the problem.

```c
char *get_name(struct Person *p) {
return p->name;
}

char *get_email(struct Person *p) {
return p->email;
}
```

```c
undefined8 get_name(long param_1)
{
return *(undefined8 *)(param_1 + 0x08);
}

undefined8 get_email(long param_1)
{
return *(undefined8 *)(param_1 + 0x16);
}
```

Fig. 3. Comparison of two structure-field accessors and their corresponding Ghidra decompilations. The decompiled functions are nearly identical, differing only in the accessed field offset (`0x08` versus `0x16`).

Second, our method is limited by the ambiguity of decompiled code. The language model re-ranker makes extensive use of the decompiled code to compare query results and correctly find the most relevant match. An issue arises when the decompiled code is very similar to the source function, but is not semantically identical because of subtle differences. In these circumstances, the re-ranker has the tendency to confirm the first incorrect candidate before considering the second. For instance, Figure 3 highlights how two functions accessing different fields of a structure can look very similar, even though the data subject is completely different. Third, our method is limited when considering functions that are represented by a limited set of anchors. Cryptographic or signal-processing kernels usually fall into this category of purely computational functions that do not interact with the operating system or use strings. The database retrieval stage struggles in such scenarios.

This work presented a practical approach for recovering source code from unknown binary functions by combining traditional reverse engineering techniques with source code search and language model-based candidate analysis. Rather than producing only decompiled pseudocode, the proposed pipeline attempts to identify the source function from a large database of available code. By extracting anchors such as strings, constants, external calls, and function names, narrowing candidate snippets, and using an LLM to reason about function similarity, the method provides a more structured and interpretable path toward binary-to-source recovery. A crucial aspect of the method is the explicit reasoning about the evidence provided by the language model, making the recovery process more interpretable by experts. The experimental results show that this approach can be highly effective when the correct source code is available in a clean and well-indexed database. In the controlled setting, the method

achieved strong matching accuracy and high instruction coverage, demonstrating that the combination of anchor-based retrieval and language-model re-ranking can recover a substantial portion of a stripped binary. The GitHub-based experiment further shows that the method can operate in a realistic and noisy environment, though its performance is currently limited by duplicated repositories and retrieval quality. Future work should focus on improving the retrieval stage, expanding and deduplicating source code databases, and incorporating stronger models or additional program analysis signals to reduce false positives. Secondary passes also offer a promising direction, since confidently matched functions can become new anchors for discovering surrounding code. This work contributes a scalable framework for source-level binary recovery, showing that reverse engineering can be significantly accelerated when static analysis, large source code corpora, and language model reasoning are combined effectively.

Disclosure of Interests. The authors declare that they have no competing interests.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, 2 edn. (2006)
2. Barr, M.: Embedded C Coding Standard. https://barrgroup.com/embedded-systems/books/embedded-c-coding-standard1 (2018), bARR-C:2018, Barr Group
3. Beninger, M., Charland, P., Ding, S., Fung, B.: ERS0: Enhancing Military Cybersecurity with AI-Driven SBOM for Firmware Vulnerability Detection and Asset Management. pp. 141–160 (2024). https://doi.org/10.23919/CyCon62501.2024.10685598
4. Cifuentes, C., Gough, K.J.: Decompilation of binary programs. Softw. Pract. Exp. **25**(7), 811–829 (1995). https://doi.org/10.1002/spe.4380250706
5. Ding, S.H.H., Fung, B.C.M., Charland, P.: Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In: Proc. IEEE S&P. pp. 472–489 (2019). https://doi.org/10.1109/SP.2019.00003
6. Dong, C., Guo, J., Yang, S., Li, Y., Fang, D., Xiao, Y., Chen, Y., Sun, L.: Advancing Binary Code Similarity Detection via Context-Content Fusion and LLM Verification. In: Proc. ASE. pp. 304–316 (2025). https://doi.org/10.1109/ASE63991.2025.00033
7. Dong, C., Li, S., Yang, S., Xiao, Y., Wang, Y., Li, H., Li, Z., Sun, L.: LibvDiff: Library Version Difference Guided OSS Version Identification in Binaries. In: Proc. ICSE (2024). https://doi.org/10.1145/3597503.3623336
8. Gagnon, C.E., Ding, S.H.H., Charland, P., Fung, B.C.M.: Beyond Embeddings: Interpretable Feature Extraction for Binary Code Similarity (2025), arXiv:2509.23449
9. GNU Project: nm: List symbols from object files (2026), https://sourceware.org/binutils/docs/binutils/nm.html, part of GNU Binutils
10. Hex-Rays: IDA Pro. https://hex-rays.com/ida-pro (2025-02-28), version 9.1
11. Li, B.: Incompressible Knowledge Probes: Estimating Black-Box LLM Parameter Counts via Factual Capacity (2026), arXiv:2604.24827

12. Li, X., Qu, Y., Yin, H.: PalmTree: Learning an Assembly Language Model for Instruction Embedding. In: Proc. ACM CCS. pp. 3236–3251 (2021). https://doi.org/10.1145/3460120.3484587
13. Massarelli, L., Luna, G.A.D., Petroni, F., Querzoni, L., Baldoni, R.: SAFE: Self-Attentive Function Embeddings for Binary Similarity (2019), arXiv:1811.05296
14. Miyani, D., Huang, Z., Lie, D.: BinPro: A Tool for Binary Source Code Provenance (2017), arXiv:1711.00830
15. Musseau, J., Meyers, J.S., Sieniawski, G.P., Thompson, C.A., German, D.: Is Open Source Eating the World's Software? Measuring the Proportion of Open Source in Proprietary Software Using Java Binaries. In: Proc. MSR. pp. 561–565 (2022). https://doi.org/10.1145/3524842.3528473
16. Nagle, F., Dana, J., Hoffman, J., Randazzo, T., Zhou, K.: Census II of Free and Open Source Software – Application Libraries. Tech. rep., Linux Foundation Research (2022). https://doi.org/10.70828/KHEH5209
17. National Security Agency: Ghidra Software Reverse Engineering Framework. https://github.com/NationalSecurityAgency/ghidra (2025-07-31), version 11.4.1
18. Pereira, D., Molloy, C., Acharya, S., Ding, S.H.H.: Automating SBOM Generation with Zero-Shot Semantic Similarity (2024), arXiv:2403.08799
19. Rahimian, A., Charland, P., Preda, S., Debbabi, M.: RESource: A Framework for Online Matching of Assembly with Open Source Code. In: Foundations and Practice of Security. pp. 211–226 (2013)
20. Tan, H., Luo, Q., Li, J., Zhang, Y.: LLM4Decompile: Decompiling Binary Code with Large Language Models. In: Proc. EMNLP. pp. 3473–3487 (2024). https://doi.org/10.18653/v1/2024.emnlp-main.203
21. Wan, B., Wang, S., Wei, Z., Huang, J., Hu, C.: Binary Code Similarity Detection via LLM-Based Source Code Conversion. IEEE Internet Things J. **12**(24), 51842–51853 (2025). https://doi.org/10.1109/JIOT.2025.3579231
22. Wang, H., Gao, Z., Zhang, C., Sha, Z., Sun, M., Zhou, Y., Zhu, W., Sun, W., Qiu, H., Xiao, X.: CLAP: Learning Transferable Binary Code Representations with Natural Language Supervision. In: Proc. ISSTA. pp. 503–515 (2024). https://doi.org/10.1145/3650212.3652145
23. Xu, H., Wang, S., Xing, Y., Feng, P., Wang, H., Li, Q., Chen, S., Sun, K.: BinProv: Binary Code Provenance Identification without Disassembly. pp. 350–363 (2022). https://doi.org/10.1145/3545948.3545956
24. Yu, S.: Accurate and Efficient SBOM Generation for Software Supply Chain Security. Ph.D. thesis, University of California, Riverside (2024)
25. Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., Wu, S.: Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. Proc. AAAI Conf. Artif. Intell. **34**(01), 1145–1152 (2020). https://doi.org/10.1609/aaai.v34i01.5466
26. Yuan, Z., Feng, M., Li, F., Ban, G., Xiao, Y., Wang, S., Tang, Q., Su, H., Yu, C., Xu, J., Piao, A., Xuey, J., Huo, W.: B2SFinder: Detecting Open-Source Software Reuse in COTS Software. In: Proc. ASE. pp. 1038–1049 (2019). https://doi.org/10.1109/ASE.2019.00100