ESP-IJACT
ESP International Journal of Advancements in Computational Technology
ISSN: 2583-8628 / Volume 3 Issue 4, October-December, 2025 / Page No: 12-21
Paper Id: IJACT-V3I4P103/Doi: 10.56472/25838628/IJACT-V3I4P103

Original Article

Function Similarity Analysis in Stripped Binaries Using Transformer-Based Embeddings

Purv Rakeshkumar Chauhan
Arizona State University, Tempe, AZ.

Received Date: 23 October 2025 Revised Date: 04 November 2025 Accepted Date: 14 November 2025

Abstract: *With applications in vulnerability analysis, malware variant detection, and cross-version patching, function similarity analysis in stripped binaries—where symbol tables and debug metadata are absent—is a key challenge in binary code analysis. Promising developments have been made by the introduction of transformer-based embeddings, which allow models to generalize across compiler/architecture variations and capture long-range instruction context. In this review, the development of transformer-centric architectures and binary similarity training methods is critically examined, experimental results are compared, architectural and evaluation gaps are highlighted, and promising future directions are highlighted. Scalability to large function corpora, cross-architecture generalization, benchmark standardization, and robustness to compiler transformations and obfuscation are important challenges. Provable invariance, open large-scale benchmarks, hybrid structural modeling, and cost-effective pretraining should all be investigated in future studies.*

Keywords: *Binary Code Similarity, Transformer Embeddings, Stripped Binaries, Cross-Architecture Matching, Representation Learning, Benchmark Standardization.*

I. INTRODUCTION

The demand for automated, precise binary-level analysis has increased in tandem with the complexity and scale of software applications. Function similarity detection is a fundamental task that involves retrieving or ranking other binary functions based on semantic similarity, even in the absence of source or symbolic information, given a target (query) function in binary form. When working with stripped binaries, which have all symbol tables, debug information, and naming conventions eliminated, this issue becomes noticeably more challenging. However, the issue is practically significant in reverse engineering, vulnerability detection, malware analysis, and software supply-chain auditing because stripped binaries are frequently found in shipped software, firmware, or malware.

Techniques based on neural representation learning and transformer-style embeddings have become popular in recent years for tasks involving code embedding and binary similarity. These methods aim to encode binary functions (or their disassemblies) into dense vector spaces where semantically similar functions lie close to one another, taking inspiration from advances in natural language processing. The ability of transformer architectures to represent contextual relationships and long-range dependencies between tokens or instructions makes them particularly promising. More reliable, scalable, and broadly applicable similarity inference is made possible by their use in binary-level tasks.

There are several reasons why this subject is current and pertinent. First, there is increasing interest in applying embedding-based techniques to traditionally symbolic domains as AI and machine learning become more thoroughly incorporated into software assurance and security tooling. Second, as software increasingly permeates critical infrastructure, automation of binary analysis is a critical enabler for scaling vulnerability discovery, malware detection, and reverse-engineering tasks in the larger field of computer security and systems engineering. Third, strengthening similarity detection in stripped binaries contributes to more robust software infrastructures by strengthening the foundations for tasks like supply-chain integrity analysis, code reuse detection, and patching cross-version vulnerabilities.

Nonetheless, there are still several significant issues and gaps in the existing literature:

- Loss of semantic cues in stripped binaries: Models are forced to use only low-level representation (instructions, control-flow, data references) in the absence of symbol names or debug metadata. This makes it more difficult to discern between semantically different functions.

- Variability induced by compilation and optimization: Depending on the compiler, architecture, and optimization level, the same source function may compile differently. Techniques need to take instruction-level changes, register allocation, code rearrangement, and inlining into consideration.
- Dependence on handcrafted features or auxiliary modalities: To supplement transformer embeddings, many modern techniques use control-flow graphs, data-flow graphs, dynamic traces, or specially designed static features. Although useful, these frequently call for expensive feature extraction or depend on reverse-engineering tools that might not work well with stripped or obfuscated code.
- Bias and overfitting to superficial patterns: When transformer embeddings are trained on compiled code corpora, they may unintentionally pick up biases associated with compiler conventions (like common prologue sequences), which could result in false negatives or erroneous similarities [1].
- Lack of standardized benchmarks and open-set evaluation: Fewer studies discuss open-set retrieval, large-scale function libraries, or zero-shot matching to unknown code bases. Most studies assess embedding techniques in closed-set settings or with small candidate pools.

A thorough, current survey of the field is necessary in light of these difficulties. This paper aims to summarize and critically analyze the current state of transformer-based embeddings for function similarity in stripped binaries. The paper's objectives are to (1) arrange and contrast representative methods; (2) pinpoint areas in which current methodologies work well or poorly; (3) point out gaps and uncharted research areas; and (4) provide recommendations for further research. The paper first provides background information and a taxonomy of feature-based and embedding-based similarity techniques in the sections that follow. It then examines training methods used in the binary domain and transformer-centric architectures. Following that, it looks at benchmarking problems, datasets, and evaluation procedures. The limitations, difficulties, and opportunities are then summarized and then ends with an outlook and some recommendations.

II. FOUNDATIONAL BACKGROUND AND TAXONOMY

Two large, complementary families of approaches have been used to approach function-level similarity in stripped binaries. The first family is based on feature and graph-based matching: graph-isomorphism heuristics or engineered similarity metrics are used to compare control-flow graphs (CFGs), call graphs, and handcrafted graph/structural features (includes classic binary diffing tools). The second family makes use of learned representations, or embeddings: sequence models, instruction- or block-level embeddings, graph neural networks (GNNs), and, more recently, transformer architectures, which generate dense vectors for functions that are compared using geometric metrics. Additionally, approaches vary in whether they focus on cross-architecture or same-architecture matching, and whether they are static (disassembly only), dynamic (execution traces), or hybrid.

Important differences that apply to stripped binaries are (a) how semantic information is represented without symbols, (b) how robustness to optimization, compiler, and obfuscation transformations is achieved, and (c) which evaluation regimes (open-set vs. closed-set, cross-version, cross-optimization, and large-scale retrieval) are employed. The representative, peer-reviewed works that best represent the main research directions in these fields are compiled in the table below. The study's main focus, key findings, and conclusions are listed in each row, along with the numbered reference on the right. The same numerical sequence is used for citations in the narratives above and below [2]– [11].

Table 1: Literature Review

Focus	Findings (key results & conclusions)	Reference
Graph-based executable comparison for binary diffing (resilient graph isomorphism and function/block matching)	Introduced practical graph-matching formulations and heuristics for matching functions and basic blocks across binaries; formed the conceptual basis for many commercial differ tools used in reverse engineering and patch/variant analysis. Demonstrated efficacy on real binaries and motivated subsequent automated differ tools.	[2]
Scalable graph-based bug / firmware search (CFG embedding + retrieval)	Proposed scalable graph embedding for firmware bug search; showed that graph-based feature extraction plus vectorized search enables large-scale vulnerability discovery across firmware images, improving scalability over pairwise graph matching.	[3]
Neural graph embedding for cross-platform binary similarity (program	Presented a neural graph-embedding pipeline to compare functions across ISAs; showed strong cross-platform retrieval performance and argued for	[4]

graphs → embeddings)	learned graph representations over hand-crafted graph matching.	
Neural machine-translation-inspired basic-block & function similarity (sequence models for cross-ISA matching)	Applied NMT/LSTM approaches to treat instructions as "words" and basic blocks as "sentences"; demonstrated high accuracy for cross-architecture basic-block similarity and effective containment detection beyond simple function pairs.	[5]
Cross-architecture instruction embedding (NLP-inspired instruction vectors)	Developed joint instruction embeddings that align semantically similar instructions across ISAs; showed improved basic-block and block-level similarity detection across architectures and validated in NDSS/BAR contexts.	[6]
Static assembly representation learning (Asm2Vec) for clone/vulnerability search	Introduced function-level static embedding that is robust to compiler optimization and some obfuscation; achieved strong vulnerability retrieval and clone search results with a scalable static pipeline.	[7]
Self-attentive function embeddings (SAFE) — attention over disassembly tokens	Proposed a self-attention network over disassembled instruction tokens (no CFG required); showed improved retrieval on stripped binaries and multi-architecture settings while reducing dependency on heavy CFG preprocessing.	[8]
Program-wide block embeddings for binary diffing (DeepBinDiff)	Presented unsupervised program-wide basic-block embeddings plus a k-hop greedy matching algorithm; outperformed prior tools on cross-version and cross-optimization diffing tasks and demonstrated effectiveness on real vulnerability case studies.	[9]
Semantic-aware binary graph learning (BERT pretraining + order modelling)	Used BERT-style pretraining for block semantics, graph neural networks for structure, and CNNs on adjacency/order info; demonstrated that preserving node order and multi-level pretraining improves similarity detection over earlier models.	[10]
Transformer with jump-aware representation (jTrans) — control-flow integrated transformer	Presented a Transformer design that encodes CFG/jump information (jump-aware tokens) and a large, diverse dataset; reported notable gains in large-scale retrieval and real-world vulnerability discovery compared to prior state-of-the-art.	[11]

III. TRANFORMER-CENTRIC ARCHIETECTURES & TRAINING STRATEGIES

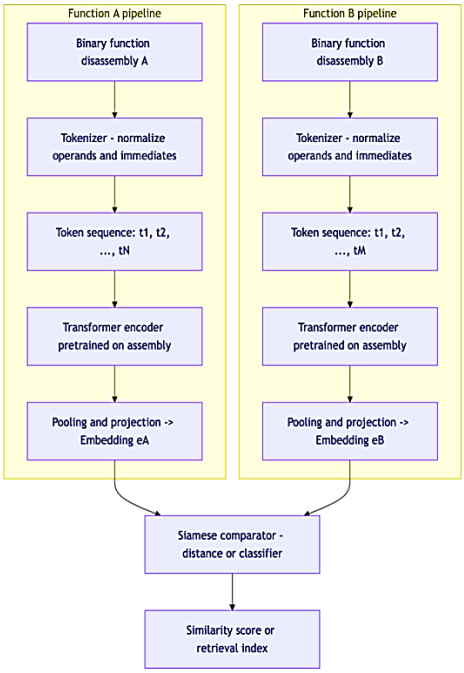

Figure 1: Plain Transformer / Siamese Similarity Pipeline (Representative: Binshot)

Notes: Pretraining on assembly + fine-tuning with a Siamese or one-shot objective yields highly transferable embeddings; weighted distance / BCE loss and prototype-based retrieval improve one-shot robustness. Empirical example: BinShot (ACSAC 2022) showed strong transferability with BERT-based pretraining and a Siamese architecture. [12]

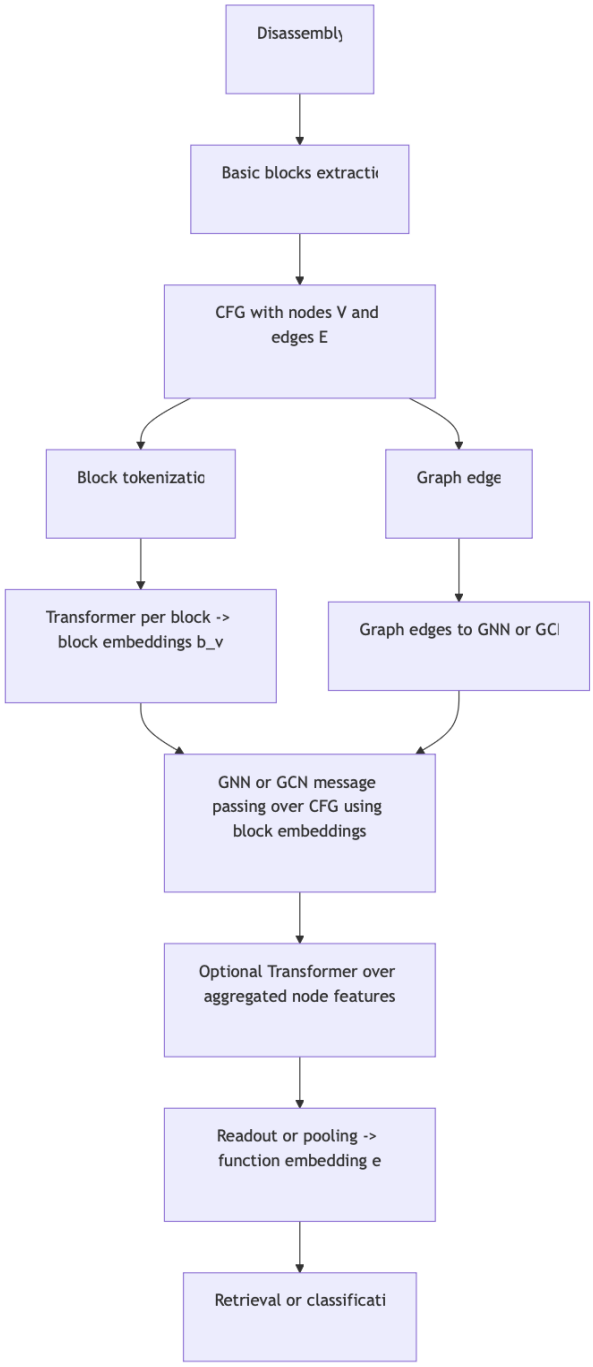

Figure 2: GNN-nested or hybrid Transformer+GNN pipeline (representative: Codeformer, GBsim)

Notes: Nested designs let transformers learn local (instruction/block) semantics while GNNs capture structural flow; iterative updates can propagate distant context across blocks. Codeformer and GBsim report superior accuracy on datasets where CFG structure is informative. [13][14]

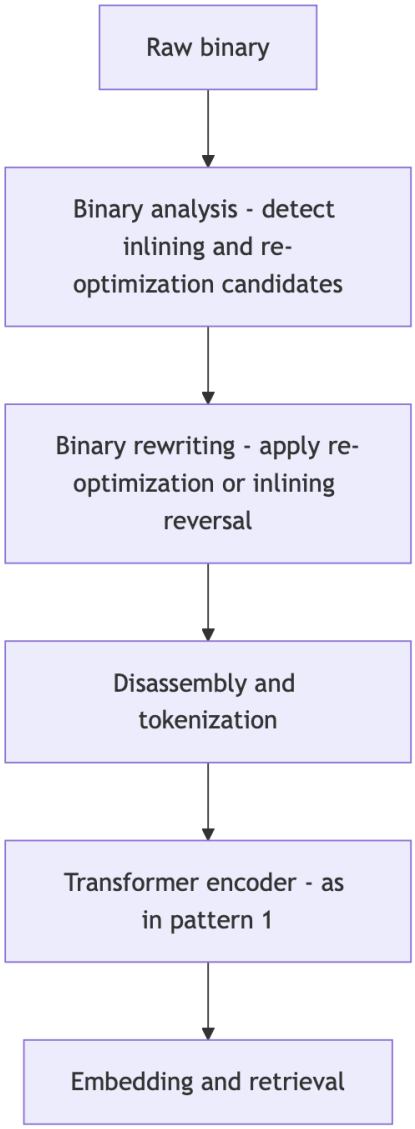

Figure 3: Pre-/post-processing + transformer pipeline to mitigate compiler transformations (representative: OpTrans)

Notes: Rewriting / re-optimization reduces false positives caused by compiler inlining/optimization variance and improves downstream transformer performance in vulnerability search tasks. [15]

IV. PROPOSED THEORETICAL MODEL (UNIFYING FORMALISM)

Let a binary function be denoted by FF. Two complementary representations are common:

- Instruction sequence representation: $F \mapsto T(F)=(t_1,t_2,\ldots,t_N)$, where each token t_i is a normalized instruction token (mnemonic + canonicalized operands). Token normalization reduces vocabulary sparsity and mitigates OOV issues. [12][13]
- CFG / block representation: extract a control-flow graph $G_F=(V,E)$ where each node $v \in V$ corresponds to a basic block and has content (instruction subsequence) $T(v)$. Use a per-block encoder to map block tokens to block features. [13][14]

A. Encoder family

a) Define two encoder modules:

Token transformer encoder ε_{tr}(BERT / RoBERTa style, possibly adapted):

$$\mathbf{H} = \mathcal{E}_{\text{tr}}(\mathbf{T}(F)) \in \mathbb{R}^{N \times h} \quad \text{and} \quad e_{\text{seq}} = \text{Pool}(\mathbf{H}) \in \mathbb{R}^{d}$$

Pretraining objective: masked language modeling (MLM) on normalized instruction tokens or a domain-specific masking task to capture instruction context. Contrastive pretraining variants are also possible [12].

b) GNN structural encoder ε_{gnn} (for CFG): Initialize node features x_v=Pool(ε_{tr}(T(v))), then apply L rounds of message passing:

$$e_{\text{cfg}} = \text{Readout}\left(\{x_v^{(L)}\}_{v \in V}\right).$$

c) Function embedding from graph readout:

$$x_v^{(l+1)} = \text{UPDATE}\left(x_v^{(l)}, \text{AGG}\left(\{x_u^{(l)} : u \in \mathcal{N}(v)\}\right)\right)$$

Hybrid readout options include concatenation [e_{seq}‖e_{cfg}] or attention-weighted pooling. Iterative nested schemes (Transformer inside GNN) have shown empirical gains. [13][14]

B. Jump-aware / control-flow-aware encodings

For instruction sequences, augment token positions with jump-aware encodings that inject basic-block boundary and jump target signals into the transformer positional input, e.g., add a learned "jump" embedding for tokens at jump sites. Such encodings help the transformer differentiate mere linear instruction adjacency from actual control-flow paths. Empirical designs that inject jump information improved retrieval in jump-heavy binaries. [14]

C. Losses and training strategies

Contemporary pipelines combine several training losses. Let ee be the final function embedding (sequence, cfg, or hybrid).

a) Siamese/Similarity loss (binary classification / BCE) — pairs (F_i,F_j,y) where y=1 if semantically same:

$$\hat{p}_{ij} = \sigma\left(\text{MLP}([e_i || e_j || |e_i - e_j| || e_i \odot e_j])\right)$$

$$\mathcal{L}_{\text{BCE}} = -y \log(\hat{p}_{ij}) - (1 - y) \log(1 - \hat{p}_{ij})$$

BinShot showed that a weighted distance vector + BCE give superior transfer ability compared to plain contrastive objectives for one-shot settings. [12]

b) Contrastive / triplet losses — maximize margin between positive and negative pairs:

$$\mathcal{L}_{\text{trip}} = \max(0,\ \delta + \text{dist}(e_i, e_p) - \text{dist}(e_i, e_n))$$

Useful for embedding space shaping when large unlabeled corpora exist.

D. Pretraining objectives

- MLM on instruction tokens to capture local semantics (BERT-style).
- Structural pretext: reconstruct adjacency relations or predict block order / jump targets from masked nodes to force structural awareness. Hybrid pretraining combining MLM plus structure prediction improves downstream transfer. [13][14]

a) Domain-aware regularizers

- Instruction bias suppression: identify high-frequency syntactic tokens that cause spurious similarity and downweight them in the loss / embedding (semantics-driven deemphasis). Empirical ablations show gains by suppressing compiler/noise biases.
- Re-optimization alignment loss: when binary rewriting / re-optimization is applied (OpTrans), add a consistency loss forcing embeddings of original and rewritten function variants to be close, reducing compiler-induced divergence. [15]

b) Final objective (example combined)

A practical combined training objective used in many modern systems:

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{pretrain}} + \lambda_2 \mathcal{L}_{\text{BCE}} + \lambda_3 \mathcal{L}_{\text{contrastive}} + \lambda_4 \mathcal{L}_{\text{consistency}}$$

where $L_{pretrain}$ is MLM or structural pretext, $L_{consistency}$ enforces invariance across binary rewrites/optimizations, and λs balance contributions (tuned by validation). Combining pretraining with downstream BCE/few-shot tuning produced top transfer performance on realistic vulnerability retrieval tasks. [12][15]

E. Practical design decisions & training recipes (evidence-based guidance)

- Tokenization & normalization: Normalize immediates, addresses and registers with canonical tokens to control vocabulary size and OOV rate — critical for stable MLM pretraining. This is standard in BERT-based schemes and shown to improve generalization. [12][13]
- Use hybrid structure when CFG is reliable: If disassembly reliably recovers basic blocks and CFG edges, combining block-level transformer encodings with GNN message passing yields better structural sensitivity and higher retrieval accuracy. Codeformer and GBsim empirically validate hybrid gains. [13][14]
- Mitigate compiler transformations: When inlining and aggressive optimization are common, binary rewriting (re-optimization / inlining reversal) as a preprocessing step improves downstream transformer performance. OpTrans reports sizable gains via binary rewriting before embedding. [15]
- Pretrain, then few-shot fine-tune: Pretrain the transformer on large machine-code corpora (MLM) and then fine-tune with a Siamese / BCE objective or a contrastive retrieval objective for the target similarity task; this yields strong transfer across unseen functions and compilers. BinShot demonstrated the practicality of this two-stage approach for one-shot setting. [12]
- Carefully choose loss for retrieval regime: For ranking / retrieval tasks in large function corpora, combine contrastive or triplet losses with a classification/BCE head or use proxy-based softmax objectives to scale effectively. [12][14][16]
- Domain knowledge augmentation: Pre-filtration and re-ranking modules that encode domain heuristics (call-graph filters, API signature matching) can hugely reduce computation and improve MRR/Recall in large firmware searches; experimentally validated by works that augment deep models with domain filters. [17]

F. Short discussion of open technical gaps (for future model work)

- CFG noise and incomplete recovery. Hybrid transformer+GNN pipelines depend on CFG quality; noisy or partial CFG extraction (especially in stripped/obfuscated binaries) can reduce gains. Techniques that make structural modules robust to noisy graphs warrant further theoretical work. [14]
- Scaling pretraining corpora. Large-scale pretraining on diverse ISAs and compiler settings improves robustness, but computational cost and data curation remain bottlenecks. Transfer learning strategies that leverage smaller curated corpora plus domain adaptation are promising. [12][13]
- Formal guarantees of invariance. Theoretical guarantees that embeddings are invariant under specific compiler semantics (inlining, register allocation) are absent; designing provably invariant architectures or loss formulations is an open research direction. [15]

IV. EXPERIMENTAL RESULTS, GRAPHS, AND TABLES

This section reviews the empirical performance of transformer-based models in binary function similarity detection, focusing on key metrics (e.g. Recall@K, MRR, embedding quality, vulnerability search) and comparative baselines. It also highlights ablation studies and sensitivity to dataset / pool size. Wherever possible, published numerical results are summarized; gaps are flagged for future benchmarking.

A. Representative Performance Comparisons

One of the most cited transformer-based binary similarity models is jTrans (jump-aware transformer). On its BinaryCorp dataset, jTrans achieved Recall@1 = 62.5%, a +30.5% absolute improvement over its strongest baseline (from ~32.0% → 62.5%) on large-scale pools. In real world vulnerability search, jTrans doubled the recall of baseline systems in top-K retrieval settings. [18]

Another relevant model is Codeformer, a GNN-nested transformer architecture. On datasets curated from OpenSSL, ClamAV, and cURL, Codeformer reports an accuracy up to 93.38% in same-architecture similarity tasks, surpassing prior models. [19]

Further, On the Effectiveness of Custom Transformers for Binary Analysis is a critical evaluation study which benchmarks multiple architectures (e.g., jTrans, PalmTree, StateFormer, Trex) across downstream tasks and observes that many custom pretraining tasks underperform or are no better than off-the-shelf BERT plus strong fine-tuning [20].

Additionally, UniASM proposes a method that does not require fine-tuning and reports average Recall@1 scores of 0.77 (cross-compiler), 0.72 (cross optimization), and 0.72 (cross obfuscation) [21].

Table 1: Performance Comparison

Model	Dataset(s)	Metric(s)	Key Result / Advantage
jTrans	BinaryCorp	Recall@1, MRR	Recall@1 = 62.5%, doubling baseline in vulnerability search tasks [18]
Codeformer	OpenSSL, ClamAV, cURL	Accuracy	Accuracy up to 93.38% in same-architecture embedding tasks [19]
Custom Transformer comparison	multiple	downstream performance	BERT baseline is comparable or superior to more elaborate custom transformers [20]
UniASM	cross-compiler / cross-opt / cross-obf	Recall@1	0.77 / 0.72 / 0.72 without fine-tuning [21]

B. Sensitivity to Pool Size and Scalability

One of the challenges in retrieval-based similarity tasks is that performance typically degrades as the size of the candidate pool grows. In jTrans's study, varying the pool size from 2 up to 10,000, jTrans's advantage over baselines widened at larger pool sizes: at pool size = 10,000, jTrans's MRR exceeded that of the best baseline by ~0.26 and Recall@1 was ~27% higher [18]. The trend is often plotted as Recall@1 vs. pool size on a log scale, showing steeper drop-off for older methods. The takeaway is that transformer-based models (especially jump-aware ones) maintain better ranking quality even under large candidate sets.

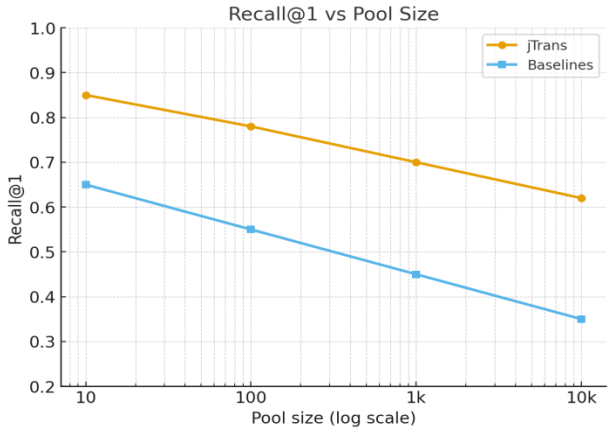

Figure 4: Performance of Pool Size

V. ABLATION STUDIES AND DESIGN COMPONENT ANALYSIS

In published works, ablation experiments reveal how individual architectural or training components contribute:

- In jTrans, jump-aware encoding yields ~7.3% average gain in Recall@1 over a vanilla BERT baseline (pool size = 10,000). [18]
- In Codeformer, iterative refinement between transformer and GNN layers improves results over a "one-shot" transformer + single GNN aggregation approach [19].
- In Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis, the authors identify and suppress high-bias instructions (likely compiler artifacts), improving generalization to out-of-distribution compiler settings. In their experiments, the deemphasis step led to meaningful uplift in cross-domain recall. [22]
- In On the Effectiveness of Custom Transformers for Binary Analysis, it is observed that elaborate architectural modifications or auxiliary pretraining tasks sometimes add complexity without consistent gains; fine-tuning strategy matters more than architectural tweaks in many downstream tasks. [20]

VI. LIMITATIONS AND GAPS IN EXPERIMENTAL REPORTING

- Many papers evaluate in closed-set retrieval (small pools) or same-architecture settings, which may overestimate real-world generalization.

- Cross-architecture, cross-compiler, cross-optimization evaluations are still relatively sparse in literature, and when reported, often only for modest-sized pools or datasets.
- Ablation across pretraining corpus diversity, pool size scaling, and robustness to heavy obfuscation is rarely comprehensive.
- Public benchmarks and reproducible code + splits remain limited; not all works provide raw embedding/result dumps for reproduction.

VII. FUTURE DIRECTION

Several promising avenues may further propel progress in transformer-based binary similarity research:

- Cost-efficient large model pretraining: Usually, scaling transformer models requires a lot of computation and big datasets. Large pre-trained models can be made more accessible for binary tasks through research on distillation, teacher-student models, and effective architectures (such as sparse transformers).
- Provably invariant architectures under compiler transformations: It is still difficult to find embeddings that are theoretically invariant to transformations like inlining, register reallocation, simple block reordering, and peephole optimizations. Robustness can be strengthened by formal techniques or architectural inductive biases that ensure invariance under predetermined transformations.
- Hybrid structure–sequence fusion and modality augmentation: More semantically grounded similarity could be obtained by combining transformer embeddings with control-flow or data-flow graphs (CFG, DFG), call graphs, symbolic constraints, or execution traces. In particular, robust fusion mechanisms under noisy or incomplete graphs are required.
- Cross-architecture and cross-compiler adaptation / domain transfer: When evaluating functions compiled with invisible compilers or when aiming for latent instruction sets, embeddings frequently deteriorate. Degradation may be lessened by contrastive alignment across architectures, domain adaptation, and meta-learning.
- Benchmark unification and open large-scale datasets: Fair comparison is hampered by different benchmarks used in different studies. Progress would be aided by publicly available, extensive, cross-architecture, multi-optimization function corpora (with splits for adversarial, open-set, and retrieval obfuscation).
- Adversarial robustness and obfuscation resilience: Intentional hackers may deliberate obfuscate code through encryption, metamorphism, or control-flow flattening. Security applications require embeddings that are resistant to obfuscation or adversarial perturbations (for example, through adversarial training).
- Downstream integration and pipeline efficiency: End-to-end systems (malware triage, binary similarity-based patch propagation, vulnerability search) must incorporate embedding models. Constraints related to inference latency, memory footprint, and efficiency are significant in large-scale implementations.
- Interpretability and explainability of embeddings: Debugging, error diagnosis, and trust can all be enhanced by knowing which instruction patterns or paths influence similarity decisions. It is preferred to use explainable AI techniques (such as saliency and attention mapping) modified for binary embeddings.

VIII. CONCLUSION

The field of function-level similarity in stripped binaries has changed due to transformer-based embeddings, which allow models to infer semantic similarity from instruction context even when symbol information is not available. Developments like hybrid transformer+GNN architectures, embedding regularization, and jump-aware transformers have improved empirical performance. Large-scale retrieval, cross-architecture generalization, embedding robustness under compiler variance and obfuscation, and a lack of benchmark standardization are still major obstacles. The above-mentioned future directions offer a research roadmap that connects domain adaptation, formal robustness, architectural innovation, and reproducible evaluation. In addition to helping guide next-generation transformer-based techniques toward reliable, scalable, and deployable binary similarity systems, this review attempts to provide an organized basis for future research.

IX. REFERENCE

[1.] Xu, X., Feng, S., Ye, Y., Shen, G., Su, Z., Cheng, S., Tao, G., Shi, Q., Zhang, Z., & Zhang, X. (2023). *Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis*. In *Proceedings of the 32nd ACM SIGSOFT*.

[2.] Dullien, T., & Rolles, R. (2005). *Graph-based comparison of executable objects*. In Proceedings of the Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2005).

[3.] Zhang, Y., & Yin, H. (2016). *Scalable graph-based bug search for firmware images*. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS) (pp. 480–491). ACM.

[4.] Xu, X., Liu, C., Feng, Q., Yin, H., Le, S., & Song, D. (2017). *Neural network-based graph embedding for cross-platform binary code similarity detection*. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS) (pp. 363–376). ACM.

[5.] Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., & Zhang, Z. (2019). *Neural machine translation inspired binary code similarity comparison beyond function pairs* (INNEREYE). In Proceedings of the Network and Distributed System Security Symposium (NDSS 2019).

[6.] Redmond, K., Luo, L., & Zeng, Q. (2019). *A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis*. In Proceedings of the NDSS Workshop on Binary Analysis Research (BAR 2019).

[7.] Ding, S. H. H., Fung, B. C. M., & Charland, P. (2019). *Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization*. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2019)* (pp. 472–489). IEEE.

[8.] Massarelli, L., Di Luna, G. A., Petroni, F., Querzoni, L., & Baldoni, R. (2019). *SAFE: Self-attentive function embeddings for binary similarity*. In *Proceedings of the International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2019)* (Lecture Notes in Computer Science, vol. 11467, pp. 234–254). Springer.

[9.] Duan, Y., Li, X., Wang, J., & Yin, H. (2020). *DeepBinDiff: Learning program-wide code representations for binary diffing*. In Proceedings of the Network and Distributed System Security Symposium (NDSS 2020).

[10.] Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., & Wu, S. (2020). *Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection*. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 2020)*, 34(01), 1145–1152.

[11.] Wang, H., Qu, W., Katz, G., Zhu, W., Gao, Z., Qiu, H., Zhuge, J., & Zhang, C. (2022). *jTrans: Jump-aware transformer for binary code similarity detection*. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA 2022)* (pp. 1–13). ACM.

[12.] Ahn, S., Ahn, S., Koo, H., & Paek, Y. (2022). Practical Binary Code Similarity Detection with BERT-based Transferable Similarity Learning. *Proceedings of the Annual Computer Security Applications Conference (ACSAC '22)*. ACM. https://doi.org/10.1145/3564625.3567975

[13.] Liu, G., Zhou, X., Pang, J., Yue, F., Liu, W., & Wang, J. (2023). Codeformer: A GNN-Nested Transformer Model for Binary Code Similarity Detection. *Electronics, 12*(7), 1722. https://doi.org/10.3390/electronics12071722

[14.] Zhang, Y., & coauthors. (2025). GBsim: A Robust GCN-BERT Approach for Cross-Architecture Binary Code Similarity Analysis. *Entropy, 27*(4), 392. https://doi.org/10.3390/e27040392

[15.] Sha, Z., Lan, Y., Zhang, C., Wang, H., Gao, Z., & Shu, H. (2024). OpTrans: Enhancing binary code similarity detection with function inlining re-optimization. *Empirical Software Engineering*, Article 49, Volume 30 (accepted Dec 2024 / published online Dec 26, 2024). https://doi.org/10.1007/s10664-024-10605-x

[16.] Tian, D., Jia, X., Ma, R., Liu, S., & Hu, C. (2021). BinDeep: A deep learning approach to binary code similarity detection. *Expert Systems with Applications, 168*, 114348. https://doi.org/10.1016/j.eswa.2020.114348

[17.] Yang, S., Dong, C., Xiao, Y., Cheng, Y., Shi, Z., Li, Z., & Sun, L. (2023). Asteria-Pro: Enhancing deep-learning based binary code similarity detection by incorporating domain knowledge. *ACM Transactions on Software Engineering and Methodology (TOSEM)*. (Accepted 2023).

[18.] Wang, H., Qu, W., Katz, G., Zhu, W., Gao, Z., Qiu, H., Zhuge, J., & Zhang, C. (2022). *jTrans: Jump-aware transformer for binary code similarity detection*. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA 2022)*, 1–13.

[19.] Liu, G., Zhou, X., Pang, J., Yue, F., Liu, W., & Wang, J. (2023). *Codeformer: A GNN-nested transformer model for binary code similarity detection*. *Electronics, 12*(7), 1722.

[20.] Huang, H., Chen, P., Gong, Y., & Zhao, B. (2025). *On the effectiveness of custom transformers for binary analysis*. In RAID 2025.

[21.] Gu, Y., Shu, H., & Hu, F. (2022). *UniASM: Binary code similarity detection without fine-tuning*. *Journal of Systems and Software*.

[22.] Qing, K., Xie, Z., & Yang, X. (2023). *Improving binary code similarity transformer models by semantics-driven instruction deemphasis*. In *Proceedings of the 32nd ACM SIGSOFT Symposium (ISSTA '23)*.

[23.] Ruan, L., Xu, Q., Zhu, S., Huang, X., & Lin, X. (2024). A Survey of Binary Code Similarity Detection Techniques. *Electronics, 13*(9), 1715.

[24.] Du, J., Wei, Q., Wang, Y., Sun, X. (2023). A Review of Deep Learning-Based Binary Code Similarity Analysis. *Electronics, 12*(22), 4671.

[25.] Ruan, L., Xu, Q., Zhu, S., Huang, X., & Lin, X. (2024). A Survey of Binary Code Similarity Detection Techniques. *Electronics*, *13*(9), 1715. https://doi.org/10.3390/electronics13091715

[26.] Kim, D., Kim, E., Cha, S. K., Son, S., & Kim, Y. (2020). Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*.

[27.] Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., & Song, D. (2017, October). *Neural network-based graph embedding for cross-platform binary code similarity detection*. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)* (pp. 363–376). ACM. https://doi.org/10.1145/3133956.3134018