REVDECODE: Enhancing Binary Function Matching with Context-Aware Graph Representations and Relevance Decoding

Tongwei Ren[1], Ronghan Che[1], Guin R. Gilman[1], Lorenzo De Carli[2], Robert J. Walls[1]
[1]*Worcester Polytechnic Institute* [2]*University of Calgary*
{tren, rche, grgilman, rjwalls}@wpi.edu, lorenzo.decarli@ucalgary.ca

Abstract

Binary reverse engineering is important for security tasks, including vulnerability discovery, malware analysis, and code reuse detection. These tasks often involve analyzing binaries without source code or debug symbols. A common yet challenging step in this process is function matching, i.e., comparing functions in unknown binaries to known reference corpora. Function matching becomes complicated due to variations introduced by differences in compilers, optimization levels, and versions. Existing matching techniques primarily focus on similarity but reverse engineers prioritize relevance—whether a match provides meaningful insights.

We present REVDECODE, a context-aware framework designed to improve function matching by leveraging interdependencies within binaries through *relevance decoding*, a technique that identifies meaningful matches based on contextual information. REVDECODE represents binaries as directed layered graphs and employs a Viterbi-inspired algorithm to determine the most relevant matches. Additionally, we propose GPU-optimized variants of REVDECODE which partition the graph traversal workload into independent subsets, maximizing GPU resource utilization and enabling greater parallelization. Experimental results demonstrate that REVDECODE significantly enhances the performance of existing function matchers, improving rankings for 56.3% to 98.8% of the evaluated functions across multiple datasets and matchers.

1 Introduction

A fundamental problem in reverse engineering is function matching, which matches functions extracted from an unknown binary to a reference corpus of known functions. This task underpins critical workflows: identifying libraries in embedded firmware [7], isolating vulnerable functions [11], and understanding code reuse in proprietary codebases [15]. The problem is complex due to variations introduced by compilation settings, optimization levels, and version differences.

Solutions to the function matching problem use a variety of techniques [2,11,25,44], including control flow structures and gadgets [2, 11], instruction and function embeddings [25, 44], and signature matching [19, 29]. Ultimately, these approaches generate abstract representations of functions and rely on *similarity* measures to identify matches. However, reverse engineers prioritize *relevance*—whether a match provides meaningful insights—over mere structural similarity. Existing methods often misclassify functionally relevant but syntactically different functions as dissimilar, creating gaps in identification, particularly in evolving codebases or under compiler-induced transformations. Moreover, current techniques struggle to distinguish relevance within ambiguous function groups that share abstract features. As discussed in Section 2.2, these challenges cannot be fully addressed by refining matching algorithms alone.

To tackle this issue, we propose *relevance decoding from context*, which identifies function relevance by leveraging surrounding contextual information. We enhance function matching by incorporating context - such as interdependencies and contextual relationships - from both individual functions and their binary environment. Relevance decoding builds on the observation that existing matchers overlook the surrounding binary context—such as preceding and succeeding code segments—which provides insights into function relationships. Our prototype, REVDECODE, employs a graph representation to capture these contextual signals and improve matching accuracy. It is not a new function-matching technique but a framework designed to enhance the accuracy of any underlying function matcher. It bridges the gap between similarity and relevance, offering a more comprehensive solution to function matching challenges.

A first challenge in our work is that matching is highly dependent on the corpus of known functions against which matching is performed. In some cases, the corpus may lack a function alltogether, or contain only an approximate match (e.g., a similar function from a different version of the same library). Thus, we forgo the goal of finding an exact match as neither useful nor productive, replacing with the notion of a set of ranked matches. In this approach, functions that bear the closest relationship with the target function are likely to

rank higher than unrelated function.

Another challenge is that incorporating context—e.g., information about functions surrounding the target function—introduces performance overhead, as it involves considering sequences of candidate functions whose combinations grow exponentially with corpus size. To address this, we propose a domain-specific graph traversal approach inspired by Viterbi decoding [40], and further enhance scalability through parallel GPU-accelerated algorithms and cascaded matching.

A final challenge lies in constructing realistic datasets of sufficient scale, particularly in domains such as firmware, where publicly available open-source samples are limited. To do so, we curate diverse datasets tailored to different analysis needs. These include open-source software libraries, synthetic yet representative *frankenbinaries*—composite binaries formed by combining functions from various libraries—and additional binaries derived from widely-used software suites.

Results show that REVDECODE is able to consistently improve function matcher ranking quality in real-world scenarios and against incomplete reference corpora, improving the rankings of 97.3% of the functions in the general-purpose dataset and between 72.3%-98.8% of the functions in the frankenbinaries dataset on average. Compared to a function matcher using solely a similarity score to compute rankings for candidate matches, REVDECODE's introduction of a context-aware direct weighted graph reduces ranking ambiguities in up to 33% of the improved function rankings in the real binaries dataset and over 50% of the improved rankings from the frankenbinaries dataset. Overall, this paper makes the following contributions:

- We introduce a context-aware ranking mechanism, integrating information about dependencies and interactions between functions. This mechanism significantly improves the reliability and interpretability of function matchers.
- We design an efficient algorithm for translating function matcher outputs into a graph with weighted edges, optimizing ranking by identifying maximum-weight paths.
- We propose GPU-optimized variants of REVDECODE which divide the work of the graph traversal into independent subsets, maximizing the utilization of GPU resources for increased parallelization. These parallel algorithms can scale to larger graph sizes while keeping end-to-end execution times under a minute.
- We evaluate our approach on an extensive dataset of binaries, showing that incorporating context can significantly improve the performance of pre-existing function matchers.

Finally, we note that REVDECODE is not designed to be robust against active obfuscation. While the use of contextual information is a double-edged sword in adversarial settings, REVDECODE remains useful in many security-focused scenarios, including stripped binary analysis, vulnerable function detection, and cross-binary comparison.

2 Background

Function matching compares functions extracted from an unknown binary to a *corpus of known functions* to measure similarities. The matcher's output is a list of *candidate functions* with similarity scores. This process enables reverse-engineering by identifying known functionality, accelerating analysis, and isolating unknown components for inspection.

Function matchers typically operate by extracting features from an unknown function and comparing them to features in the corpus. These tasks often lack the support of source code, debugging information, or documentation, necessitating effective and efficient automated techniques.

2.1 Measuring Function Similarity

Function matchers employ diverse techniques to measure similarity. For each category we provide, if applicable, a few examples; we review additional approaches in Section 8.

- **Control Flow Analysis:** Tools such as BSim [1] and discovRE [11] analyze control flow structures to identify similarities between functions. These approaches use graph-based representations, such as Control Flow Graphs (CFGs), and compute similarity using methods like Maximum Common Subgraph Isomorphism or feature vector comparisons.
- **Machine Learning Techniques:** SAFE [25] and Gemini [44] use deep learning models to generate embeddings for functions. These embeddings encode high-level semantic and structural information, enabling similarity comparisons through vector operations such as cosine similarity.
- **I/O Behavior and Signature-Based Matching:** Pewny et al. [28] derive signatures based on I/O behavior for bug identification. Similarly, BinHash [19] and related approaches focus on input-output behaviors at the basic block level.
- **Statistical and Heuristic Features:** Although not common in literature, it is in principle possible to use simple heuristics to reduce functions to sets of features, such as instruction mnemonics or basic block behaviors. Similarity can be computed using metrics like the Jaccard coefficient.

2.2 Challenges in Existing Function Matching

Similarity versus Relevance: Function matchers primarily focus on measuring similarity between functions, typically leveraging syntactic or structural features like control flow graphs or instruction sequences. Nonetheless, similarity does not necessarily imply relevance for reverse engineers, who prioritize grasping the functional intent and operational context of the code [41]. Low similarity scores do not always indicate a lack of relevance, as functions may exhibit significant changes in their implementation while retaining the same functional purpose. For instance, consider a scenario where a

reverse engineer encounters an unknown function, which is actually version 2.40 of `foo`, but only version 2.0 of `foo` is known. Significant source code changes between these versions may add new logic or modify existing functions, which could lead the function matcher to label the two versions of foo as dissimilar. Nevertheless, the reverse engineer still needs to recognize that the code corresponds to some iteration of foo to track its evolution and assess its impact. Thus, the challenge lies in bridging the gap between similarity-based approaches and the relevance-driven needs of reverse engineers.

Handling Incomplete Corpora: Building a comprehensive corpus of all possible functions is infeasible. Consider the libc library: it has countless versions, each potentially compiled for different target architectures, using varying compilers, build options, applied patches, and so on. As a result, it is almost certain that any given binary will contain variants of known code rather than exact matches. This require the ability to match in the presence of such differences, but also to avoid returning a match when the uncertainty is too great.

Existing matchers are designed to be robust to some degree of variation, but none of the tools we tested provide mechanisms to label a function as *uncertain*. The quality of function matching should not overly depend on the completeness of the corpus, as building such a complete corpus is unrealistic.

Ambiguity in Matches: Ambiguous matches represent the converse of the incomplete corpus issue. As the corpus grows, the likelihood of encountering highly similar functions increases. For instance, hardware abstraction layer (HAL) libraries across different platforms tend to share similar function structures, making them difficult to distinguish.

Existing matchers struggle to differentiate these functions because they rely on features that are insufficient to discriminate between such cases. This forces analysts to manually resolve ambiguities, which significantly increases the time and effort required for analysis.

2.3 Leveraging Contextual Information

Our insight is that existing function matchers underutilize contextual information from the corpus and binary. Contextual information from the corpus includes the uniqueness of particular features. For example, matching a rare feature might indicate a higher likelihood of a correct match.

Contextual information from the binary involves leveraging strong matches in one part of the binary to inform the matching process in others. If a strong match identifies a known library in one section of the binary, this information can guide matching for the surrounding functions.

3 Design Of REVDECODE

In this section, we propose a function-matching optimization framework that attempts to address the issues outlined in Section 2 by conceptualizing function matching across an entire binary as a directed layered graph, where layers represent unknown functions, nodes in the graph represent candidate functions, and edges and weights on those edges encode contextual relationships. We call our implementation of this framework REVDECODE.

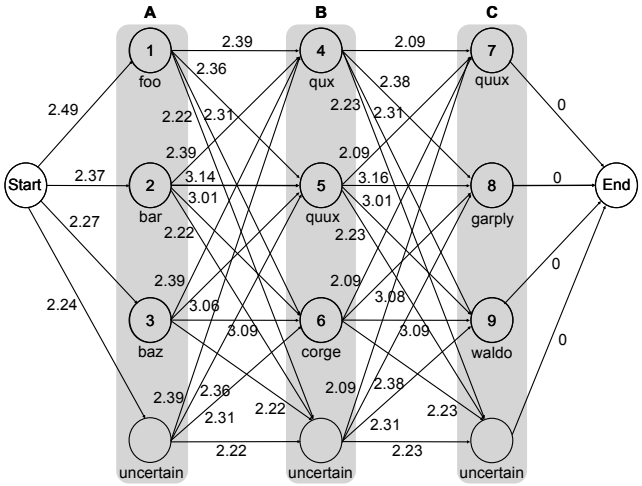
REVDECODE does not replace function matchers; rather, it is a system that incorporates results from existing function matchers. The input to REVDECODE is an unknown binary and a reference corpus of functions. REVDECODE then constructs a graph (discussed in the following), using similarity scores returned by the function matcher, as one component of the weight of edges between function nodes. REVDECODE then efficiently finds the maximum-weight path through the constructed graph. The final ranking for each unknown function is based on the proximity of the candidates to that path.

REVDECODE consists of three primary phases. First is the graph construction phase, which builds the graph based on the unknown binary and the reference corpus. Second is the graph traversal phase, where the system identifies the most relevant paths based on edge weights and contextual relationships. Third is the ranking phase, which creates a ranked list of candidates for each function in the unknown binary.

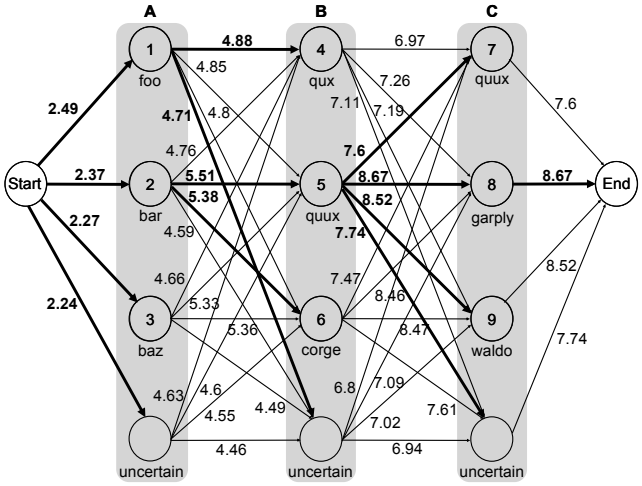
3.1 Graph Construction

Candidate functions from the reference corpus are represented as nodes in a directed acyclic graph (DAG). Each node corresponds to a potential match for an unknown function from the input binary. The nodes are arranged into layers, with one layer dedicated to each unknown function. Thus, given n unknown functions, each unknown function U_j where $j \in \{1,2,...,n\}$, becomes a layer in the graph, consisting of a set of candidate matches M_i where $i \in \{1,2,...,m\}$, with m being the total number of candidate matches for U_j. Additionally, REVDECODE adds three special nodes types: start, end, and uncertain nodes. The start and end nodes represent the start and end points for path traversal, meaning there is only one node of each of these types in the constructed graph. The uncertain node type allows REVDECODE to be robust to incomplete corpora, and there is one uncertain node included per layer. We describe them further in Section 3.1.2 below.

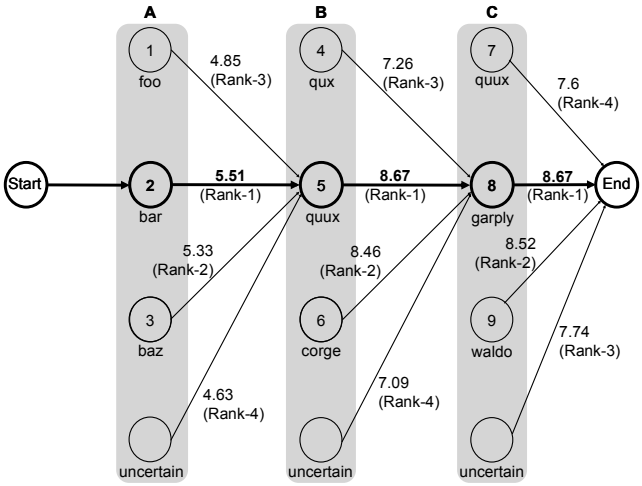
REVDECODE constructs edges between nodes to represent adjacency relationships between unknown functions in the input binary. If two unknown functions are adjacent in memory within the binary, they are modeled as adjacent layers within the graph. Further, for every pair of consecutive unknown functions U_j and U_{j+1}, an edge $e_{i,j,k}$ is added between every candidate match M_i in layer U_j and every candidate match M_k in layer U_{j+1}. This design leverages the observation that the compilation process frequently places functions from the same compilation unit contiguously in memory, encoding this contextual information in the graph. Additionally, REVDECODE places the start node before the first layer, containing



(a) Constructed directed acyclic graph.



(b) Graph after calculating the weight for each path. Bolded edges denote the maximum incoming path of each dest. node.



(c) Graph after ranking. Bolded nodes are the nodes falling on maximum-weight paths.

Figure 1: A simple example to show REVDECODE.

an outgoing edge to every node in that layer. Similarly, the end node is placed after the last layer.

An example of a constructed graph can be seen in Figure 1. Visually, the graph layers are arranged from left to right according to the unknown functions' offsets in memory. Each layer corresponds to a single unknown function. A candidate function from the corpus may correspond to multiple nodes in the graph if those nodes are in separate layers.

We can represent the graph as $G = (V, E)$, where $V = \{v_s, v_e\} \cup \{v_{i,j} | i \in \{1, ..., m+1\}, j \in \{1, n\}\}$. Here, v_s, v_e are respectively the start and end nodes, and the node $v_{i,j}$ represents the $i-th$ candidate match for the $j-th$ unknown function in layer j. Each layer has an additional uncertain node represented as $v_{m+1,j}$.

The set of edges is defined as $E = E_S \cup E_L \cup E_E$. E_S represents the set of edges connecting node v_s to all nodes in layer 1. E_E represents the set of edges connecting all nodes in layer n to v_e. $E_L = \{(v_{i,j}, v_{k,j+1}) | i, k \in \{1, ..., m+1\}, j \in \{1, n-1\}\}$ is the set of directed edges between layers, where each edge connects a candidate match node $v_{i,j}$ in layer j to a node $v_{k,j+1}$ in layer $j+1$, representing transitions between adjacent unknown functions in the binary.

3.1.1 Edge Weights

Each edge $e_{i,j,k}$ in E has an associated weight $w_{i,j+1}$ which are computed as the summation of four distinct scores, with each score taking a value between 0 and 1. Higher values indicate stronger potential matches.

Similarity Score. A measure of similarity between the unknown function and the candidate at the destination node, leveraging existing function-matching algorithms (e.g., BSim, discoveRE, etc.). The choice of algorithm is configurable and we evaluate the various options in Section 6. REVDECODE normalizes the raw similarity scores using Sigmoid normalization [6]. Sigmoid normalization emphasizes small differences in scores near the center of the range while suppressing extreme values, mitigating the impact of outliers.

Confidence Score. A metric quantifying how many features are shared between an unknown function and a candidate at the destination node, weighted by their uniqueness relative to the corpus. It is derived from the Term Frequency-Inverse Document Frequency (TF-IDF) statistical measure [20]. This score highlights features that are common between the two functions but rare across the entire corpus. The calculation for this score is provided in Appendix 12.2.1. We again use Sigmoid normalization to normalize confidence scores.

Adjacency Score. A value that quantifies the contextual relationship between two candidate functions connected by an edge. The score calculation is shown in Appendix 12.2.2. The score increases if both candidates originate from the same library, and is incrementally boosted if they also share the

same version and optimization level. Additionally, if the candidates are from the same compilation unit within that library, the score is further increased.[1] We set the baseline score (0.7) using a dataset of 30 synthetic firmware samples and maximizing the cumulative Discounted Cumulative Gain (DCG).

Library Score. A measure of the uniqueness of a candidate's library based on its contribution to the overall corpus. The calculation is shown in Appendix 12.2.3. Libraries with fewer functions relative to the total corpus will have higher library. Conversely, libraries with a higher proportion of functions in the corpus will have lower library scores, reflecting their broader prevalence and reduced specificity. Functions from the same library receive the same library score.

3.1.2 Uncertain Nodes

REVDECODE adds a special node in each layer to encode a notion of uncertainty. This uncertainty represents two key ideas. First, the corpus may be incomplete, meaning it might not contain a relevant match for the unknown function. Second, candidates should have contextual support beyond the similarity score to provide evidence they are relevant. In other words, the uncertain nodes in each layer allow REVDECODE to return *uncertain* in the final rankings. Any candidates ranked above *uncertain* have contextual evidence in the binary.

To set the edge weights for the uncertain nodes, REVDECODE matches the similarity score of the uncertain node to the maximum of the other candidates in the layer. The confidence score is set to 85% (configurable) of the self-confidence before normalization. Self-confidence refers to the confidence score of an unknown function if it were matched against itself, i.e., every feature matched. The uniqueness score is set empirically using a training set. We used a value of 0.8 in our evaluation. The adjacency score is set to zero, as the uncertain function is not related to any corpus candidate functions.

3.2 Graph Traversal: Weight Computation

The goal of the graph traversal phase is to find the maximum weight path through the graph, which can then be used in the next phase to create a set of rankings of candidate functions for each unknown function, i.e., each layer in the graph. This phase consists of a forward pass through the graph which can be viewed as a variation of the Viterbi algorithm [40] with notable differences. REVDECODE employs a graph-based construction rather than a hidden Markov model and computes path weights instead of symbol emission and transition probabilities, as described below. Also, in comparison to the Viterbi backward pass, REVDECODE aims to identify all nodes that lie on or in close proximity to at least one of the maximum weight paths rather than identifying a specific path.

[1] In the current implementation, we approximate compilation unitis using DWARF information during corpus construction, treating functions as belonging to the same unit if they come from the same source file.

The forward pass iteratively constructs a weight matrix, starting from the start state and progressing through the layers. Every edge $e_{i,j,k}$ in the graph has an entry in the matrix that represents the total weight of the maximum weight path beginning at v_{start} and ending with $e_{i,j,k}$. The weight matrix computation follows an iterative dynamic programming approach to solve the recurrence relation, which executes in polynomial time. The best incoming cumulative weight $W(v_{i,j})$ for node $v_{i,j}$ is calculated as:

$$W(v_{i,j}) = \max_{v_{k,j-1} \in V} [W(v_{k,j-1}) + w(v_{k,j-1}, v_{i,j})], \quad (1)$$

where $W(v_{k,j-1})$ is the best cumulative weight for node $v_{k,j-1}$ in previous layer $j-1$, and $w(v_{k,j-1}, v_{i,j})$ is the weight of the edge from $v_{k,j-1}$ to $v_{i,j}$. For the nodes in layer 1, the best cumulative weight is calculated as:

$$W(v_{i,1}) = W(v_{start}) + w(v_{start}, v_{i,1}), \quad (2)$$

where $W(v_{start})$ is initialized to 0. Finally, for the end node v_{end}, the best cumulative weight is calculated as:

$$W(v_{end}) = \max_{v_{i,n} \in V} [W(v_{i,n}) + w(v_{i,n}, v_{end})] \quad (3)$$

3.3 Graph Traversal: Ranking

After the graph traversal phase generates the weight matrix for the edges in the graph, the matrix is used to rank candidate nodes at each layer based on their proximity to the maximum weight path. This is accomplished by a backward pass through the graph, starting at the end node and leveraging the weight matrix to determine which nodes lie on one of the maximum weight paths to the start node.

Starting from the end node, candidate nodes in the last layer n are ranked by the cumulative weights of their outgoing edges to the end state: $rank(v_{end}) = sorted(W(v_{i,n}) + w(v_{i,n}, v_{end}) | v_{i,n} \in V)$. This ranking represents the nodes' proximity to the maximum weight path.

We then define the set of rank one nodes in layer n as $R(n)$. The node with the highest weight is ranked first and is guaranteed to be on a maximum weight path; in cases where multiple nodes are tied for the maximum cumulative weight, they all receive rank one status. The rank one nodes from this layer effectively become the new end states, and the ranking process is repeated for all previous layers.

For each preceding layer j, we then collect the rankings $rank(v_{i,j+1})$ from each end state (rank one node) in the previous iteration, $v_{i,j+1} \in R(j+1)$. Each ranking is therefore a list of groups of nodes in layer j sorted by their cumulative weights, as each of the rank one nodes from layer $j+1$ has a distinct set of incoming edges from the previous layer, forming individual rankings per rank one node.

To derive the final ranking for a layer j, these individual rankings are combined. For each rank r, the nodes within that rank from each individual ranking are merged to form the final ranking, with the highest rank achieved in any ranking determining the final rank. Thus, the new set of rank one nodes for layer j is given by the merged rank one nodes.

This process is repeated for all layers $j = n, n-1, ..., 2$. This guarantees that all nodes on all maximum weight paths are found, and that the nodes in each layer are ranked by their proximity to the maximum weight paths. Furthermore, by making the rank one nodes the new end states at each step, we ensure that both that nodes which are not on a maximum-weight path are excluded from the backtracking, and that all nodes' rankings are refined at each step based on the discovery of each node in a maximum weight path.

The output of this step is a set of matches for each candidate function, ranked by relevance as computed above.

3.4 Cascaded Matching

Function matching effectiveness improves as the reference corpus grows, since larger corpora increase the chance of including relevant candidate functions. However, we empirically observe that some matchers - particularly those based on modern ML techniques [25, 44] - incur a significant computational cost, with throughput below 1K matches/s even on modern hardware (ref. Section 6.1.2). For those, computing similarity score of each unknown function against the entire corpus quickly becomes unpractical as the corpus grows.

To enable these computation-intensive matchers to scale to realistic corpus sizes, REVDECODE leverages an optional phase which we term *cascaded matching*. If cascaded matching is enabled, REVDECODE first uses a fast matcher (typically one based on syntactic or heuristic matching, such as BSim [1]) to pre-generate an initial ranking of the top-250 candidate functions for each unknown function. Then, the computation-intensive matcher is only applied to these candidates, filtering out the remaining ones from the corpus. REVDECODE then constructs the directed layered graph using similarity scores from the intensive matcher, followed by the graph traversal and ranking phases.

A drawback of this approach is that it causes computation-intensive matchers performance to partially depend on those of a different matcher (the one used in the pre-filtering). This issue is however mitigated by the fact that even a fast matcher will typically rank highly relevant functions within the top-250. Thus, in practice, we found the performance impact to be minimal, as demonstated in our evaluation (Section 6.1.2). On the other hand, pre-filtering enables ML-based computationally-intensive techniques to be brought to bear on the problem of binary function matching; thus, we consider the trade-off to be acceptable.

4 GPU-Accelerated REVDECODE

We propose parallel algorithms for REVDECODE's graph traversal to overcome scalability challenges posed by large binaries and candidate sets. These algorithms preserve the semantics of the original ranking procedure while supporting high-throughput matching. This section presents the parallelization strategies and their GPU-based implementation.

4.1 Challenges in Parallelizing REVDECODE

Designing parallel versions of graph traversal and incorporating those into REVDECODE presents challenges:

Sequential Dependencies. The computation of each layer depends on the results of the previous one, preventing naive parallelization across layers.

Resource Utilization. GPUs divide work between *thread blocks*, which each independently execute subsets of the work in parallel; threads within each block perform the same set of instructions on multiple pieces of data simlutaneously. In order to take advantage of the large number of available threads, the work for the forward pass phase must be divided efficiently among thread blocks to maximize the amount of work being done in parallel.

Memory Dependence. The relatively simple computations and frequent accesses to the cumulative weight matrix create potential memory bottlenecks. The memory access patterns of the threads must be carefully designed to ensure that they do not generate contention for memory resources that slows down the runtime of the kernel.

To address these challenges, we propose two variants of REVDECODE with parallelized traversal algorithms: *fine-grained traversal* and *segment-based estimation traversal*. These variants aim to optimize parallel execution of the forward pass phase while minimizing memory contention across GPU streaming multiprocessors (SMs). They differ primarily in their approach to forward pass construction.

4.2 Fine-Grained Traversal

The key insight of fine-grained traversal is to add parallelism by assigning each GPU thread a single edge transition between candidates across consecutive layers. Each candidate node per layer is assigned a separate thread block, and each thread within that block computes the weights for one incoming edge. The graph is processed sequentially at the layer level, yet within each layer, computations are parallelized.

The forward pass stage for fine-grained traversal goes layer by layer, launching a set of thread blocks for each node within the layer that each compute their weight calculations in parallel. For each thread block, each thread computes the cumulative weights for its assigned edge:

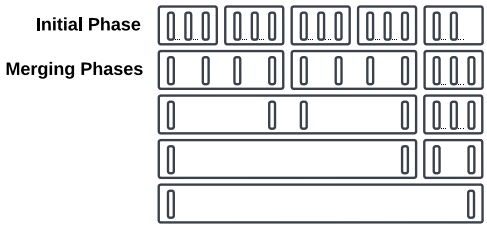


Figure 2: Illustration of segment-based estimation traversal. The graph is initially divided into five segments, which are merged pairwise in successive stages.

$$C_{k,i} = W(v_{k,j-1}) + w(v_{k,j-1}, v_{i,j}) \tag{4}$$

Then, threads within a block use a parallel sorting algorithm, bitonic sort [5], to sort the cumulative weights $C_{k,i}$ into descending order. The bitonic sort is computed at this stage instead of in a backward pass stage because the threads in each thread block, having been assigned to one particular edge in the graph, are perfectly positioned to perform the sorting computations in parallel.

Thus, the thread with the highest cumuluative weight will end up at the first position, $C_{0,i}$, and this corresponds to the maximum cumulative weight for $v_{i,j}$, $W(v_{i,j})$. The maximum path is then given by the corresponding predecessor node, and the rankings for $v_{i,j}$ are determined by the sorted order of the cumulative weights.

4.3 Segment-Based Estimation Traversal

Segment-based estimation traversal employs an approximation strategy to boost scalability, sacrificing a bit of accuracy for better performance with larger binaries. The fundamental idea is to implement a divide-and-conquer approach, breaking the graph into several segments, with each segment representing a series of consecutive layers within the graph. One thread block is used for each segment, and these results are then merged, as illustrated in Figure 2.

This approach is based on the assumption that the maximum weight paths calculated in each segment act as an approximation for the optimal paths of the entire graph. The segment size is picked such that there is at least one thread block per SM whenever possible, maximizing parallelization and the utilization of GPU resources. The initial stage and merging process are described in detail below.

4.3.1 Initial Stage

Each thread block processes its allocated segment of the state transition graph independently and concurrently. Inside each segment, threads within the block sequentially compute cumulative weights for $v_{k,j-1}$ for each previous layer $j-1$ within the block's segment:

$$C_{k,i} = W(v_{k,j-1}) + w(v_{k,j-1}, v_{i,j}), \forall v_{k,j-1} \in V_{block} \tag{5}$$

Each thread then ranks the cumulative weights $C_{k,i}$ by sorting them in descending order, and then obtains the maximum cumulative weight.

4.3.2 Merging Stage

In the following phases, adjacent segments are merged to incorporate dependencies that span segment boundaries. During each merging phase, pairs of adjacent blocks merge their boundary computations together, gradually integrating the cumulative weight data for the graph and reducing the number of layers and of active thread blocks at each step. For each pair of adjacent blocks, we focus on boundary unknown functions, ensuring that cumulative weights reflect transitions from prior segments: the *left boundary node* is the last unknown function U_{jL} of the left block, while the *right boundary node* is the first unknown function U_{jR} of the right block.

There are then two distinct cases for merging boundary nodes based on their adjacency in the original graph.

Adjacent Boundary Nodes. When U_{jL} and U_{jR} were adjacent in the original graph, we perform full forward pass computations, calculating the cumulative weights by considering all of the predecessor nodes in the left boundary of U_{jL}, and updating the ranking and maximum path selection.

Non-Adjacent Boundary Nodes. When not adjacent in the graph, only the cumulative weight update is calculated by combining weights from the left boundary:

$$W(v_{i,jR}) = W(v_{i,jR}) + W_{left}, \tag{6}$$

where W_{left} represents the maximum cumulative weight from the left boundary. The rankings and maximum paths are not updated.

After each merging phase, GPU-wide synchronization ensures updates to cumulative weights and paths are fully propagated and visible across segments.

4.4 Data Structures and Memory Layout

For both parallel variants, REVDECODE organizes the cumulative weight matrix structure to align with the order in which GPU thread warps will access it. Each weight $w(v_{k,j-1}, v_{i,j})$ is modeled as a 3D tuple with entries for: **Rows:** The number of candidates in the current unknown function U_j. **Columns:** The number of unknown functions, corresponding to the sequence of functions being matched. **Depth:** The number of candidates in the preceding unknown function U_{j-1}.

The transition weights between two consecutive unknown functions are stored along the column (layer) dimension. The weights are stored in *depth-major order*, meaning that for

each candidate in the current unknown function, REVDECODE consecutively stores the weights of transitions from all candidates in the previous unknown function. By arranging the weights in this way, REVDECODE enables coalesced memory accesses during the forward pass computations. This reduces the amount of memory congestion that would otherwise be generated by the large number of accesses to the cumulative weight array performed by the threads during that phase of the calculations.

5 Evaluation Methodology

To evaluate the effectiveness of REVDECODE, we developed a methodology that includes both real-world binaries and synthetic firmware-like samples. Our evaluation focuses on ranking quality using tie-aware NDCG as the primary metric. We constructed two datasets: one consisting of general-purpose binaries compiled from widely used open-source projects, and another comprising synthetic embedded binaries designed to model the structure and diversity of firmware. We generated ground truth using debug information and manual verification, selected four representative matchers to compute similarity scores, and applied additional controls to address sources of bias, such as function duplication in shared libraries. We describe these components in detail below.

5.1 Dataset: General Purpose

To evaluate REVDECODE's ranking performance across a variety of real-world binaries, we assembled binaries from three widely-used projects—GNU Binutils, BusyBox, and OpenSSL—compiled across multiple Ubuntu releases and versions. In all cases, we aimed to recreate each release's build environment so that the resulting binaries closely resemble those found in practice. We used debug symbols to establish ground truth, though they were hidden from REVDECODE during matching. We also employed manual verification checks of the ground truth and simple data sanitization to handle compiler-introduced changes, such as name mangling.

In total, the dataset consists of 132 binaries containing 262,486 functions, split between the reference corpus and the evaluation set. The reference corpus includes 68 binaries with 54,821 functions, while the evaluation set comprises 64 binaries with 207,665 functions. Together, these produce over 11 billion function pairs for evaluation.

GNU Binutils Suite. We collected all Binutils versions available in currently supported Ubuntu releases (14.04 through 24.10), spanning Binutils 2.24 up to 2.43.1. This includes the core utilities (`ld`, `as`, `objdump`, etc.) as well as their associated libraries—most notably Libbfd, Libc (2.19–2.40), and Libz. Each utility was compiled with the toolchain and package versions appropriate to its Ubuntu release, using link-time optimization (LTO) and static linking for evaluation binaries. This setup allows us to assess ranking performance on both program-specific functions (the standalone utilities) and shared library functions embedded via static linking.

BusyBox. BusyBox combines dozens of standard Unix commands into a single, compact executable, making it common in embedded and low-footprint Linux systems. We built ten BusyBox versions, from 1.23.3 through 1.37.0, each using its the build environment from the corresponding Ubuntu release. The single binary structure of BusyBox provides an interesting case study for REVDECODE's context encoding, as each command's functions originate from a shared binary image but are conceptually separate.

OpenSSL. OpenSSL's libssl and libcrypto libraries implement a suite of cryptographic primitives and protocols. Unlike Binutils and BusyBox, OpenSSL libraries are not standalone executables but are linked into other binaries. We included four releases: 3.0.2, 3.0.8, 3.2.1, and 3.5.0.

Reference Corpus and Evaluation Set. For evaluation, we divided the dataset into a reference corpus and an evaluation set. The reference corpus includes Binutils binaries from Ubuntu releases up to 20.04, along with their associated libraries; BusyBox versions 1.23.3, 1.26.2, and 1.32.1; and OpenSSL version 3.0.2. The remaining binaries—i.e., those not included in the reference corpus—served as the evaluation set—served as the evaluation set. To better simulate challenging real-world scenarios, we compiled the Binutils evaluation binaries as statically-linked files with link-time optimization (LTO) enabled. Due to limitations in the BSim implementation provided by Ghidra, we were unable to obtain results for 13 binaries, as listed in Appendix **??**. To maintain consistency and fairness across all matchers, we excluded these binaries from the evaluation set for every matcher.

5.2 Dataset: Frankenbinaries

To evaluate REVDECODE in the context of embedded systems, we created a separate dataset of synthetic firmware samples. While some public firmware datasets exist, they lack the detail required to establish reliable ground truth. To address this, we constructed custom binaries, referred to as *frankenbinaries*, that simulate key structural properties of real firmware. These include the number, type, and layout of libraries and functions commonly found in embedded environments.

Each frankenbinary is composed according to a fixed set of constraints. Functions and libraries are selected uniformly at random, subject to the following rules: each function must contain at least ten instructions; all functions from a given library are laid out sequentially in memory; no library appears more than once per binary; and each binary includes exactly one hardware abstraction layer (HAL) library.

To populate the frankenbinaries, we curated a collection of open-source libraries targeting the widespread ARMv6/7-M architectures. The library set spans multiple applica-

tion domains and includes HAL libraries for STM32 microcontrollers, cryptographic libraries such as OpenSSL and MbedTLS, networking components like Mosquitto, the FreeRTOS real-time operating system, and additional libraries such as OpenCV, CycloneDDS, and Aubio. All libraries were compiled using GCC 9.2, under optimization levels ranging from `-O0` to `-O3` and `-Os`, to introduce compiler-induced variation. A complete list of libraries and versions is provided in Appendix 12.3. In total, this dataset included 168 libraries containing 107,634 functions.

Reference Corpus and Evaluation Set. Rather than relying on a single corpus for this dataset, we generated six separate corpora to simulate different forms of incompleteness. The `all_versions_all_opts` corpus includes every available binary variant for all libraries. In contrast, corpora with names containing `*O2_opt*` restrict the corpus to library binaries compiled with the `-O2` optimization level. Similarly, the `latest_version*` and `oldest_version*` corpora retain only the most recent or earliest version of each library, respectively. For the evaluation set, we synthesized 300 frankenbinaries comprising a total of 26,389 functions.

5.3 Measuring Ranking Quality

To assess the ranking quality of REVDECODE's outputs, we adopted tie-aware Normalized Discounted Cumulative Gain (NDCG) as the primary evaluation metric. NDCG is well-suited to the function matching problem because it accounts for both the order and relevance of candidate functions.

We assigned relevance scores to candidate functions based on their potential utility to a reverse engineer. An exact match received the highest score of 15. Partial matches—such as the same function compiled under different options or sourced from a different version—were scored between 9 and 3, depending on the degree of similarity. In cases where no appropriate candidate was present in the corpus, we assigned the uncertain label a non-zero relevance score. The algorithm that shows the specific assignment of relevance scores is provided in Appendix 12.2.4.

Often, multiple candidates in the corpus carry a non-zero relevance for a given unknown function. An ideal ranking should place these candidates in strict order of their relevance scores. We also note that REVDECODE operates without access to these ground-truth scores.

5.4 Additional Considerations

Limitations of Other Datasets. We explored several alternative datasets, but found them to be insufficient for our evaluation for various reasons. The dataset from Marcelli et al. [24] lacks labels needed for identifying functions across binaries. VarCorpus [27] lacks library-level context. The FirmXRay [43], Shannon Firmware [3], and Fuzzware [35] datasets lacked sufficient ground truth.

Matcher Implementations. We selected four existing matchers to provide similarity scores: BSim, discovRE, SAFE, and Gemini [44]. For SAFE and Gemini, we used the implementations provided by Marcelli et al [24]. Since the source code for discovRE was not publicly available, we reimplemented it based on the design and heuristics described in the original publication. For BSim, we used the implementation bundled with Ghidra version 11.0.

Handling Function Duplication. A common challenge in function matching evaluation is the duplication or overrepresentation of functions, e.g., those originating from widely used shared libraries [27]. This function duplication can artificially inflate performance metrics. To mitigate this effect, we performed additional analyses that control for potential duplication. These include evaluation subsets that exclude functions from known shared libraries, as well as subsets that consider only functions that have undergone significant changes across versions.

Hardware. We ran CPU-based computations on a server with two Intel Xeon Gold 5218 CPUs (32 logical cores, 503 GB RAM) and GPU-based computations on an NVIDIA A100-SXM4-80GB GPU (108 SMs, 2,048 threads per SM, 1,024-thread block size).

6 Evaluation

We evaluate REVDECODE in terms of both runtime performance and ranking effectiveness. The first subsection analyzes the system's efficiency and scalability, while the second examines how REVDECODE improves the quality of function rankings by integrating contextual information.

6.1 Runtime Performance

We evaluate REVDECODE's runtime performance along three key dimensions: overall end-to-end performance, the impact of cascaded matching on matcher scalability, and the efficiency of the GPU-accelerated graph traversal algorithms. These results demonstrate that REVDECODE delivers high-quality rankings with runtimes that make it practical.

6.1.1 End-to-End Performance

Across the 12 utilities in GNU Binutils v2.34—drawn from the evaluation set and comprising a total of 37,005 functions (median: 3,099 per binary)—REVDECODE with GPU-accelerated fine-grained graph traversal completed analysis in an average of 7 minutes and 10 seconds per binary. These function counts include both utility-specific code and statically linked library functions.

The most time-consuming phase was graph construction, primarily due to the cost of computing similarity scores using the BSim matcher. This phase took an average of 5 minutes and 23 seconds per binary, making similarity computation the dominant performance bottleneck. In contrast, the forward and backward graph traversal phases combined took just 16 seconds per binary on average, highlighting the efficiency of the proposed fine-grained parallel traversal algorithm. This result aligns with expectations for binaries containing around 3,000 unknown functions and 251 candidates per function (c.f., Sec. 6.1.3).

An additional 91 seconds per binary was spent on miscellaneous operations such as file I/O. Much of this overhead reflects prototype-level logging and artifact generation, which would be unnecessary in a production deployment.

The use of the more computationally intensive SAFE matcher for computing similarity scores remains tractable when paired with cascaded filtering, adding just 6 minutes and 58 seconds per binary. Without this optimization, SAFE would require an estimated 23 hours per binary.

Overall, these results demonstrate that REVDECODE's architecture enables timely and scalable analysis, with overall performance primarily constrained by the efficiency of the underlying similarity score computation.

6.1.2 Scalability and Cascaded Matching

REVDECODE's optional cascaded matching phase enables the use of computationally intensive matchers, such as SAFE and Gemini, without incurring prohibitive runtime costs. When used naively, computing similarity scores across all function pairs in the general-purpose binary dataset would require more than 79 days for SAFE and over 211 days for Gemini. In contrast, REVDECODE's cascaded matching reduces this burden by introducing a fast, first-stage filter that narrows the candidate set before invoking slower matchers.

By applying BSim as the initial filter, the number of function pairs was reduced by a factor of 200, resulting in a dramatic reduction in the total number of matches, from billions down to 52 million. BSim was chosen for its high throughput (540,000 matches per second) and competitive accuracy relative to other fast matchers. For comparison, DiscovRE processes approximately 250,000 matches per second, while SAFE and Gemini process just 1,600 and 600 matches per second, respectively.

This reduction translated directly into tractable runtimes. Matching all function pairs with BSim alone required approximately 10 hours; DiscovRE, evaluated without cascading, took 22 hours. For SAFE and Gemini, the cascade approach proved crucial: the BSim filtering stage took 10 hours, after which SAFE completed second-stage matching in 9 hours and Gemini in 23 hours. Without cascading, these matchers would be impractical for large-scale analysis.

This performance gain comes with minimal impact on the quality of the results. BSim included high-relevance matches in 93% of the target functions after the first stage. Further analysis revealed that the 7% without a high-relevance match had only a minor impact on the final ranking performance in the second stage. When we artificially added the missing high-relevance matches to the candidate sets to compute an upper bound on the impact, SAFE's NDCG improved from 0.68 to just 0.69, and the percentage of functions with improved rankings rose modestly from 88.0% to 89.3%. Gemini demonstrated nearly identical performance under both conditions, with 94.4% and 94.3% of improved rankings, respectively, and an average NDCG of 0.73 in both cases.

6.1.3 Scalability and Graph Traversal

Figure 3 compares the execution times of three graph traversal strategies—*naive traversal*, *fine-grained traversal*, and *segment-based estimation traversal*—across progressively larger graphs generated from our frankenbinaries. Naive traversal serves as a baseline, launching a single GPU thread block and walking the graph layer by layer, assigning one thread to each candidate node in the current layer. Although fully parallel within a layer, it performs no additional optimizations and therefore reflects the upper bound on what a straightforward GPU traversal algorithm can achieve.

Fine-grained traversal maintains a runtime below 50 seconds for every tested graph size, delivering more than a 5x speedup over naive traversal for the largest graphs (501 candidates per function) and an over 40x speedup relative to a simple CPU implementation we profiled on the same inputs. Segment-based estimation traversal is faster still: it cuts the forward-pass time by an average of 3.77x compared with fine-grained traversal, and reduces end-to-end traversal time by 1.39x. The modest improvement in total runtime is because matrix initialization and transfer to global memory account for over half of the total runtime.

The additional speed of segment-based estimation traversal comes at almost no cost in ranking quality. On the Binutils dataset, segment-based estimation traversal achieved an NDCG of 0.719 versus 0.722 for fine-grained traversal. Segment-based estimation traversal inherently trades off accuracy for scalability with the number of segments, where a larger number of segments will increase parallelizability at the cost of requiring more estimations of optimal paths. However, the amount of lost accuracy is dependent on both the available hardware resources and the size of the graph, as these factors together determine the segment size. The number of segments is chosen to maximize the utilization of hardware resources. In this case, there are 512 candidate matches (and threads) per unknown function, and 432 segments (and thread blocks). This allows for all SMs and all 2,048 threads per SM to be occupied. As the Binutils utilities consist of thousands of functions, each segment encompasses a large enough portion of the graph that there was little impact on accuracy overall.

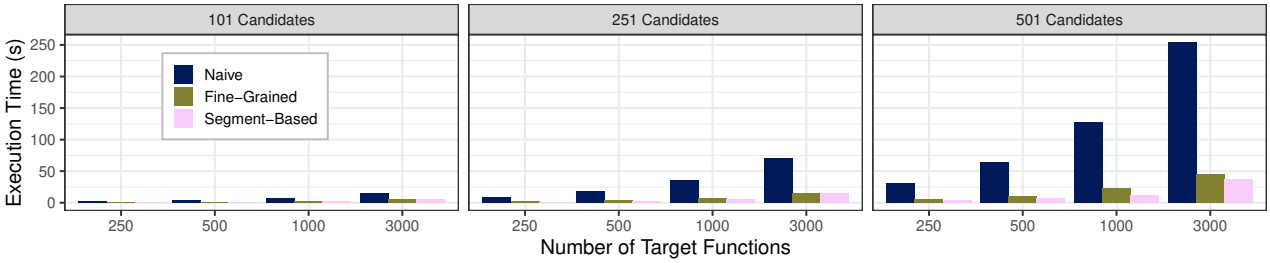


Figure 3: Execution times of the naive, fine-grained, and segment-based estimation graph traversal algorithms.

Both optimised algorithms remain constrained by GPU resources. Fine-grained traversal requires one thread per incoming edge and segment-based estimation traversal one thread per node, capping the maximum size at 1,024 candidates per unknown function on GPUs that limit thread-block size to 1,024. Likewise, the weight matrix for all layers must fit in global memory; larger graphs could be supported through oversubscription techniques such as NVIDIA Unified Virtual Memory, provided effective page-eviction and prefetching policies are in place [4, 13].

These results demonstrate that REVDECODE's parallel traversal stage can analyze graphs with thousands of functions and hundreds of candidates per function in seconds, transforming traversal from a potential bottleneck into a negligible fraction of total runtime. The design scales gracefully within current GPU limits, and modest hardware advances—or careful memory management—promise even greater capacity without sacrificing accuracy.

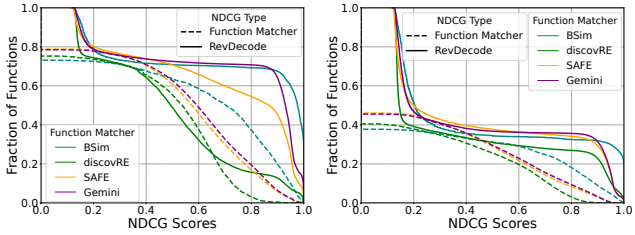
6.2 Ranking Effectiveness

Below, we evaluate how REVDECODE refines matcher outputs to produce high-quality rankings. First, we present macro-level improvements across two datasets. We then examine scenarios where REVDECODE is especially effective, identify its key limitations, and conclude with an ablation study that highlights the contribution of each contextual signal.

6.2.1 Macro-Level Ranking Improvements

REVDECODE significantly improves the quality of function rankings produced by existing matchers across both the general-purpose and frankenbinary datasets.

General-Purpose Dataset. As shown in Figure 4a, REVDECODE offers substantial improvements in ranking quality across all evaluated matchers. For BSim, 97.3% of the 207,665 rankings improved, resulting in an average NDCG increase from 0.55 to 0.75. SAFE saw improvements in 88.0% of rankings, with the average NDCG rising from 0.50 to 0.68. Gemini improved 94.4% of rankings, raising the NDCG from 0.51 to 0.73. DiscovRE had 56.3% of rankings improved, and NDCG rose from 0.43 to 0.49.



(a) All functions. (b) Without shared libs.

Figure 4: NDCG scores of rankings derived from raw similarity scores (dashed lines) versus rankings produced by REVDECODE using the same matchers (solid lines), evaluated on the general-purpose dataset.

General-Purpose Dataset (Excluding Shared Library Functions). As shown in Figure 4b, REVDECODE continued to deliver marked improvements even after excluding functions from shared libraries. Among the 87,837 remaining functions, BSim rankings improved in 97.1% of cases, raising average NDCG from 0.27 to 0.46. SAFE improved 93.9% of rankings (NDCG from 0.27 to 0.45), Gemini 93.0% (from 0.27 to 0.45), and discovRE 94.5% (from 0.22 to 0.39).

Frankenbinary Dataset. As shown in Table 1, REVDECODE also improves ranking performance on synthetic firmware samples. Across all matchers, the percentage of improved rankings ranges from 72.3% to 98.8%.

6.2.2 Strengths

REVDECODE demonstrates particular strength in two high-impact scenarios: (1) recovering relevant matches that are structurally dissimilar due to version or compilation differences, and (2) resolving ambiguity among functions that lack distinctive features. In both cases, REVDECODE's ability to incorporate contextual cues from the surrounding graph enables it to recover high-quality rankings that standalone similarity scores fail to identify.

Recovering Structurally Divergent Matches. Function matchers often struggle when the target function has undergone significant structural changes compared to its counterparts in the corpus. This scenario arises frequently across library versions and compiler configurations, leading to low

similarity scores and poor ranking performance for even highly relevant matches. REVDECODE excels in these cases by leveraging the graph structure and contextual signals from nearby strong matches.

For example, consider the function `custom_exts_copy` from `libssl-3.5.0` in the general-purpose dataset. DiscovRE, operating on raw similarity scores, incorrectly identifies `elf32_arm_size_stubs` from `libbfd-2.30` as the best match, while the actual high-relevance match—`custom_exts_copy` from `libssl-3.0.2`—is buried at rank 197. REVDECODE successfully promotes this match to rank 1 by using neighboring high-relevance, high-similarity matches.

A similar case arises with BSim, which misidentifies `uncompress` from `libz` (Ubuntu 20.04) as `get_absolute_expression` from `as-2.24`. REVDECODE again corrects the ranking, promoting the correct high-relevance match from rank 187 to the top position.

To quantify this behavior, we analyzed a subset of 2,568 functions in the general-purpose dataset for which high-relevance matches existed in the corpus but were missed by BSim at rank 1 due to structural changes. REVDECODE improved 1,994 of these cases. Of those, 900 involved promoting a true high-relevance match, with 761 achieving the top rank. The remaining 1,094 improvements were due to better ranking of lower-relevance but still meaningful matches (e.g., same library, different function).

Compiler-induced variation presents another form of structural divergence. In the frankenbinaries dataset, for instance, BSim fails to rank the most-relevant version of `get_zeros_padding` from `MbedTLS-2.15.1` (compiled with `-Os`) among the top candidates when matching against a corpus compiled with `-O2`, assigning it rank 114. However, surrounding functions like `ecp_double_jac`, `mbedtls_internal_aes_decrypt`, and `mbedtls_cipher_write_tag` are structurally similar to their corpus versions. REVDECODE exploits this context to elevate the correct match to rank 1.

Resolving Ambiguity Among Low-Signal Functions. A second area where REVDECODE excels is in disambiguating functions that exhibit weak or common feature sets. These include short functions with few instructions as well as functions composed of common code idioms. These cases typically lead matchers to produce high similarity scores for candidates that are not relevant.

For example, `generic_bignum_to_int64` from `as-2.38` triggered 250 different rank-1 matches when using BSim. This ambiguity stems from its short, undifferentiated structure. Prior work has often addressed this by excluding such functions entirely; for instance, discovRE filters out any function with fewer than five basic blocks, and the SAFE and Gemini implementations from Marcelli et al. [24] apply similar heuristics.

Rather than discarding these challenging cases, REVDECODE leverages contextual relevance to recover meaningful rankings. In the general-purpose dataset, 70,720 of the 202,058 improved BSim rankings (35%) fall into this category. In the frankenbinaries dataset, nearly half of all improvements stem from resolving ambiguity in structurally ambiguous functions, including many from embedded HAL libraries that share common implementation patterns.

These results highlight REVDECODE's ability to preserve ranking quality in cases that typically confound existing matchers, extending its utility beyond easy matches to precisely the cases where automated assistance is most needed.

6.2.3 Weaknesses

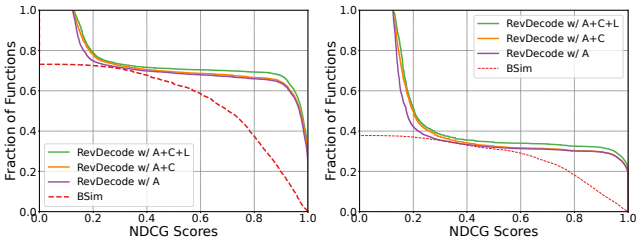
While REVDECODE substantially improves ranking quality across a broad range of scenarios, its effectiveness is constrained by certain inherent limitations. In particular, two key challenges remain: the absence of relevant matches in the corpus and the persistence of residual ambiguity when contextual signals are indistinguishable.

No Relevant Matches in the Corpus. REVDECODE, like the underlying function matchers it relies on, cannot recover a relevant match if one does not exist in the corpus. This limitation is fundamental to similarity-based approaches in reverse engineering and arises when functions in the evaluation set are entirely novel or structurally distinct from anything previously seen.

Consider the function `unw_decode_uleb128` from `readelf-2.38`. This function does not appear in any earlier version of `readelf` included in the corpus. As a result, BSim's top 250 candidates do not contain any relevant matches. Instead, the rankings are dominated by structurally similar functions from unrelated binaries, such as `busybox` and `objdump`.

REVDECODE attempts to mitigate this limitation by introducing the *uncertain* label into the ranking. However, this strategy is not foolproof. In this case, because unrelated `busybox` and `objdump` functions also appear in the BSim results for adjacent unknown functions, REVDECODE interprets these as contextual support and elevates them in the ranking for `unw_decode_uleb128`. This phenomenon, where unrelated functions form reinforcing chains of incorrect context, can lead to systematic ranking errors. We refer to this effect as *irrelevant context chains*.

Residual Ambiguity in Indistinguishable Contexts. Another limitation arises when functions exhibit not only similar features but also indistinguishable contextual relationships. In such cases, REVDECODE cannot disambiguate candidates any more effectively than the raw similarity score allows. For example, the function `cmd_SignatureAlgorithms` from `libssl` remains ambiguous even after applying REVDECODE. Using discovRE as the matcher, there are four distinct matches from `libssl-3.0.2` at rank 1, each receiving the



(a) All functions. (b) Without shared libs.

Figure 5: NDCG scores for REVDECODE variants with different scoring components. Rankings are derived using only Adjacency (**A**), Confidence (**C**), and Library (**L**) scores, or combinations thereof.

same contextual score due to identical similarity, adjacency, and library scores.

Although these unresolved ambiguities represent a much smaller fraction of the total compared to those observed when using similarity scores alone, they highlight an inherent ceiling on REVDECODE's disambiguation capability when no unique contextual signal is available.

The above limitations suggest directions for future work. Addressing irrelevant context chains may require more sophisticated weighting or confidence estimation of contextual signals. Meanwhile, resolving *residual ambiguity* could benefit from additional sources of semantic or structural information beyond what is currently encoded in the graph.

6.2.4 Ablation Study

Role of Contextual Signals. REVDECODE's ability to produce high-quality rankings stems from the way it integrates multiple contextual signals—similarity, adjacency, confidence, and library affinity—into its graph-based reasoning. Each of these signals plays a role in shaping how relevance propagates through the graph, and together they enable REVDECODE to recover meaningful matches even when similarity alone falls short.

Figure 5 highlights the cumulative impact of these signals on the general-purpose dataset. Even with only similarity and adjacency information, REVDECODE lifts BSim's average NDCG from 0.55 to 0.72. Adding the confidence score offers modest improvements, raising the NDCG to 0.73. Finally, incorporating the library score increases NDCG to 0.75. For the subset with shared library functions removed, shown in Figure 5b, REVDECODE improves BSim's NDCG from 0.27 to 0.42 with just similarity and adjacency, then to 0.44 with confidence, and finally to 0.46 with library information.

Role of the Uncertain Node. REVDECODE's *uncertain* node plays an important role in mitigating irrelevant context chains. When no strong candidate emerges in a given layer, the highest weight path will pass through the uncertain node, which, by design, neither contributes nor receives contextual bonuses. This halts further propagation of weak or spurious evidence, preventing unrelated functions from reinforcing each other across layers.

Adding the uncertain node led to a decrease in context-chain-induced ranking errors: roughly 7% overall, and nearly 10% on the set without shared library functions. While the uncertain node cannot recover a relevant match where none exists, it helps ensure that poor evidence remains isolated.

7 Discussion

Opportunities and Challenges for Future Work. While REVDECODE mitigates the challenges posed by incomplete corpora, corpus quality remains a critical factor in function matching performance. This work does not focus on corpus construction; however, our evaluation provides valuable insights that inform effective corpus design strategies.

Insights from handling compiler options suggest that corpus construction should emphasize aggressive optimization levels such as `-O3`, and `-Os`. These introduce significant structural changes, posing greater challenges for matchers.

Conversely, to improve scalability, the corpus should avoid redundant functions that exhibit minimal differences from existing versions. Eliminating such redundancy can help manage corpus growth more efficiently.

To further enhance scalability, adopting a hierarchical corpus structure may prove beneficial. Matching could proceed in successive passes through progressively detailed layers of the hierarchy. Higher layers should be smaller, more diverse, and optimized for triage, while lower layers can be larger and tailored to specific binary types, such as a reference corpus for STM32-based firmware libraries.

Broader Security Applications of Function Matching. Numerous security workflows rely on the ability to identify semantically equivalent functions across different binaries. This capability is particularly critical when dealing with stripped binaries, where symbols and debug information are absent. In such scenarios, matching unknown functions to a database of known ones can reduce reverse engineering effort, accelerating tasks such as vulnerability and malware analysis. Function similarity also plays a vital role in identifying and patching vulnerabilities in statically linked third-party libraries. Other downstream applications include binary diffing, software lineage analysis, and automated malware clustering.

The prevalence of these use cases is evidenced by the wide body of research on binary function similarity across systems security, machine learning, and programming languages communities. In many related works [24, 25, 29, 44], these applications are highlighted as primary motivations for developing new matching algorithms. REVDECODE contributes directly to the effectiveness of these security applications by enhancing function matching performance.

Matcher	Corpus	Imp.	Deg.	Unch.	Mean Orig.	Med. Orig.	Mean RevD.	Med. RevD.	Mean Imp.	Med. Imp.	Mean Deg.	Med. Deg.
BSim	corpus_all_Ver_all_Opt	23125	2959	305	0.93	0.94	0.97	0.98	0.06	0.04	-0.02	-0.02
BSim	corpus_all_Ver_O2_Opt	19832	6440	117	0.89	0.94	0.95	0.98	0.10	0.06	-0.07	-0.06
BSim	corpus_latest_Ver_all_Opt	21737	4164	488	0.93	0.96	0.99	0.99	0.07	0.04	-0.01	-0.00
BSim	corpus_latest_Ver_O2_Opt	19099	6618	672	0.90	0.95	0.97	0.99	0.12	0.07	-0.06	-0.02
BSim	corpus_oldest_Ver_all_Opt	21370	4928	91	0.92	0.96	0.97	0.99	0.08	0.04	-0.06	-0.04
BSim	corpus_oldest_Ver_O2_Opt	19089	7226	74	0.89	0.95	0.95	0.99	0.12	0.06	-0.10	-0.10
SAFE	corpus_all_Ver_all_Opt	25989	369	31	0.91	0.93	0.98	0.99	0.07	0.05	-0.11	-0.04
SAFE	corpus_all_Ver_O2_Opt	22565	3785	39	0.87	0.93	0.92	0.99	0.08	0.05	-0.09	-0.03
SAFE	corpus_latest_Ver_all_Opt	22067	3883	439	0.91	0.96	0.97	0.99	0.07	0.04	-0.04	-0.00
SAFE	corpus_latest_Ver_O2_Opt	23216	2660	513	0.89	0.96	0.92	1.00	0.06	0.02	-0.14	-0.07
SAFE	corpus_oldest_Ver_all_Opt	23425	2785	179	0.91	0.95	0.96	0.99	0.06	0.03	-0.08	-0.01
SAFE	corpus_oldest_Ver_O2_Opt	23719	2640	30	0.89	0.95	0.93	1.00	0.06	0.03	-0.12	-0.07
discovRE	corpus_all_Ver_all_Opt	26081	189	119	0.90	0.92	0.98	0.98	0.08	0.05	-0.13	-0.02
discovRE	corpus_all_Ver_O2_Opt	21230	5119	40	0.83	0.91	0.87	0.98	0.09	0.06	-0.14	-0.09
discovRE	corpus_latest_Ver_all_Opt	24606	1525	258	0.90	0.94	0.96	0.99	0.08	0.04	-0.13	-0.01
discovRE	corpus_latest_Ver_O2_Opt	21981	4353	55	0.84	0.93	0.88	0.99	0.08	0.04	-0.17	-0.13
discovRE	corpus_oldest_Ver_all_Opt	24562	1629	198	0.89	0.94	0.95	0.99	0.07	0.05	-0.16	-0.05
discovRE	corpus_oldest_Ver_O2_Opt	22022	4350	17	0.84	0.92	0.89	1.00	0.09	0.05	-0.15	-0.11
Gemini	corpus_all_Ver_all_Opt	25530	760	99	0.89	0.92	0.96	0.97	0.08	0.05	-0.13	-0.07
Gemini	corpus_all_Ver_O2_Opt	21896	4441	52	0.83	0.91	0.86	0.97	0.07	0.04	-0.18	-0.13
Gemini	corpus_latest_Ver_all_Opt	23565	2557	267	0.89	0.94	0.94	0.99	0.07	0.04	-0.13	-0.03
Gemini	corpus_latest_Ver_O2_Opt	22413	3911	65	0.85	0.93	0.88	0.99	0.07	0.04	-0.19	-0.15
Gemini	corpus_oldest_Ver_all_Opt	23984	2219	186	0.88	0.93	0.93	0.99	0.07	0.05	-0.17	-0.10
Gemini	corpus_oldest_Ver_O2_Opt	23178	3186	25	0.85	0.92	0.90	1.00	0.09	0.06	-0.17	-0.12

Table 1: NDCG comparison on the frankenbinaries dataset: rankings derived from raw similarity scores vs. rankings produced by REVDECODE. **Imp.**, **Deg.**, and **Unch.** indicate the number of rankings that improved, degraded, or remained unchanged. **Mean/Med. Orig.** and **Mean/Med. RevD.** report NDCG scores before and after applying REVDECODE. **Mean/Med. Imp.** and **Mean/Med. Deg.** show the average and median magnitude of NDCG improvements and degradations.

8 Related Work

Function Matching. We summarize additional function matching techniques that we excluded from evaluation due to incompatibility with REVDECODE's model or inferior performance. HALucinator's LibMatch [7], for example, returns only one match per function and cannot support ranked retrieval. BinHunt [14] relies on graph isomorphism and symbolic execution, making it prohibitively expensive for large codebases. Pewny et al. [28] and BinHash [19] also use control-flow-based features, which we already evaluate through DiscovRE [11]. BinDNN [21] uses LSTMs for function matching. Genius [12] and the approach by Zuo et al. [45] embed basic blocks using learned representations. These embedding-based methods have been surpassed by SAFE and Gemini in prior comparisons.

Binary Code Fingerprinting. Prior work has proposed fingerprinting techniques to trace malware provenance [18], detect unauthorized code reuse in proprietary firmware [15], distinguish firmware from other binary artifacts [9], or identify known vulnerabilities [10, 22]. These approaches typically operate at the whole-binary level. In contrast, REVDECODE focuses on matching individual functions within a binary. Similar in spirit to REVDECODE, FirmUp [10] fingerprints known vulnerable functions by taking into account their surrounding context using model theory. However, FirmUp targets identification of specific known functions and does not scale to matching across an entire binary. REVDECODE, by contrast, leverages contextual information to support function-level matching at scale.

Identification of Bugs. Another body of work [8, 30, 37] use program analysis—including symbolic execution, taint analysis, or custom static analyses—to identify previously unknown security issues. While orthogonal to our focus, function matching could complement these efforts by recovering semantically related functions.

Source-level code fingerprinting. A substantial body of work addresses code clone detection at the source level [16, 17, 26, 36, 38]. These approaches generally leverage program features that are lost or obfuscated during compilation. While such features could, in principle, be principle recovered via decompilation, doing so is a fragile and error-prone process [23].

Forensic Triage. Our work is inspired partly by past work [39, 42] on forensic triage. However, their goals, design choices, and application domains differ substantially from REVDECODE. Whereas prior systems were tailored to tasks like smartphone investigation and relied on manually guided analyses, REVDECODE targets scalable function-level matching for reverse engineering. It constructs a directed graph with weighted edges derived from ranking data to encode contextual information. Relevance scores are inferred automatically based on ground truth, eliminating the need for investigator input.

9 Conclusions

Existing approaches to function matching, which rely on structural similarity, often misclassify functionally relevant but syntactically different functions, leading to incomplete identification in evolving codebases.

To address these limitations, we proposed REVDECODE, a relevance decoding framework that enhances function matching by integrating context within binaries to refine match assessments, improving reliability and interpretability.

Evaluation demonstrates that REVDECODE significantly enhances function matcher performance by reducing ambiguities and improving ranking accuracy. In real-world scenarios, it improves the rankings of 97.3% of functions in general-purpose dataset and up to 98.8% in frankenbinary dataset. By systematically applying relevance decoding, REVDECODE bridges the gap between similarity and relevance, offering a practical solution to the challenges of function matching.

10 Open science

In adherence to the USENIX Security open science policy, the following artifacts are available to the public: an implementation of REVDECODE with sample binaries and instructions for running the code [33]; the general purpose dataset used in the evaluation, including the evaluation set and the reference corpus set [31]; the 300 frankenbinaries along with the BSim matching results [34]; and the NDCG scores used to draw Figures 4a, 4b, 5a and 5b [32].

11 Ethics Considerations

This section explores ethical considerations in terms of respect for persons, beneficience, and justice.

Based on common community definition, the work in this paper does not constitute *human subject research*. In accordance with this consideration, we did not identify any potential immediate harm to individuals from our work, thus *respect for persons* is not a concern.

As for *beneficience*, the broad purpose of our work lies in improving capabilities of binary reverse engineering, which is of benefit to software consumers, developers, and security practitioners. We utilize a mix publicly available external software datasets, and our own internally generated dataets. As for evaluated function matchers, we use either publicly available author-provided code, or our own reimplementation based on publicly-available publications. This approach will enable us to publicly release our code and datasets to foster greater transparency and reproducibility in research.

Finally, concerning *justice*, our research did not involve including or excluding entities based on attributes of a given group of persons (as this is not relevant to our study).

Acknowledgments

We thank the anonymous reviewers and shepherd. We also thank Heshan Perera, Joshua DeOliveira, and Daniel Reynolds for their contributions to this project. This material is based upon work supported by the National Science Foundation under Award No. 2154415.

References

[1] Ghidra. https://ghidra-sre.org, 2019.

[2] Introduction to bsim. https://github.com/ NationalSecurityAgency/ghidra/blob/master/GhidraDocs/GhidraClass/BSim/BSimTutorial_Intro.md, 2024.

[3] Shannonbaseband. https://github.com/grant-h/ShannonBaseband, 2025.

[4] Tyler Allen and Rong Ge. In-depth analyses of unified virtual memory system for gpu accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[5] K. E. Batcher. Sorting networks and their applications. *IEEE Transactions on Computers*, C-20(12):1465–1471, 1968.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[7] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218. USENIX Association, August 2020.

[8] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. Pie: Parser identification in embedded systems. In *Proceedings of the 31st Annual Computer Security Applications Conference*.

[9] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Towards automated classification of firmware images and identification of embedded devices. In *ICT Systems Security and Privacy Protection*. Springer International Publishing, 2017.

[10] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[11] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. 02 2016.

[12] 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, page 480–491, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 224–235, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Debin Gao, Michael K. Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In Liqun Chen, Mark D. Ryan, and Guilin Wang, editors, *Information and Communications Security*, pages 238–255, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[15] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *IEEE Working Conference on Mining Software Repositories*, 2011.

[16] Takashi Ishio, Yusuke Sakaguchi, Kaoru Ito, and Katsuro Inoue. Source file set search for clone-and-own reuse analysis. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 257–268, 2017.

[17] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *ICSE*, pages 96–105, 2007.

[18] Steven Jilcott. Scalable malware forensics using phylogenetic analysis. In *2015 IEEE International Symposium on Technologies for Homeland Security (HST)*, 2015.

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

[20] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *J. Documentation*, 60:493–502, 2021.

[21] Nathaniel Lageman, Eric D. Kilmer, Robert J. Walls, and Patrick D. McDaniel. Bindnn: Resilient function matching using deep learning. In Robert Deng, Jian Weng, Kui Ren, and Vinod Yegneswaran, editors, *Security and Privacy in Communication Networks*, pages 517–537, Cham, 2017. Springer International Publishing.

[22] 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.

[23] Zhibo Liu and Shuai Wang. How far we have come: testing decompilation correctness of c decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[24] 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.

[25] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. *Lecture Notes in Computer Science*, page 309–329, 2019.

[26] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 153–164, New York, NY, USA, 2018. Association for Computing Machinery.

[27] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, and Ruoyu Wang. "len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4069–4087, 2024.

[28] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724, 2015.

[29] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. *it - Information Technology*, 59, 01 2017.

[30] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher

Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *IEEE Symposium on Security and Privacy*, 2020.

[31] Tongwei Ren, Ronghan Che, Guinevere Gilman, Lorenzo De Carli, and Robert J. Walls. Revdecode : General purpose dataset. https://doi.org/10.5281/zenodo.15579566, June 2025.

[32] Tongwei Ren, Ronghan Che, Guinevere Gilman, Lorenzo De Carli, and Robert J. Walls. Revdecode : Ndcg scores. https://doi.org/10.5281/zenodo.15580030, June 2025.

[33] Tongwei Ren, Ronghan Che, Guinevere Gilman, Lorenzo De Carli, and Robert J. Walls. Revdecode: Code. https://doi.org/10.5281/zenodo.15588740, June 2025.

[34] Tongwei Ren, Ronghan Che, Guinevere Gilman, Lorenzo De Carli, and Robert J. Walls. Revdecode: Frankenbinary evaluation dataset. https://doi.org/10.5281/zenodo.15581150, June 2025.

[35] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *USENIX Security*, 2022.

[36] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *SIGMOD*, page 10, 2003.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Network and Distributed System Security Symposium*, 2015.

[38] Randy Smith and Susan Horwitz. Detecting and Measuring Similarity in Code Clones. In *IWSC*, page 7, 2009.

[39] Saksham Varma, Robert J. Walls, Brian Lynn, and Brian Neil Levine. Efficient smart phone forensics based on relevance feedback. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, page 81–91, New York, NY, USA, 2014. Association for Computing Machinery.

[40] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

[41] Daniel Votipka, Seth Rabin, Cristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. USENIX Association, 2020.

[42] Robert J. Walls, Erik Learned-Miller, and Brian Neil Levine. Forensic triage for mobile phones with dec0de. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, page 7, USA, 2011. USENIX Association.

[43] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 167–180, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 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, page 363–376, New York, NY, USA, 2017. Association for Computing Machinery.

[45] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS)*, 2019.

12 Appendix

12.1 Graph Size

As described in Section 3.1, REVDECODE constructs a directed graph where each layer corresponds to an unknown function and each node represents a candidate match. As the number of candidates increases, the graph grows larger, leading to higher computational costs. To reduce the number of candidates in each layer, REVDECODE applies an initial filtering phase. Only the top candidate matches with the highest similarity scores are retained. A candidate match from the corpus may correspond to multiple nodes in the graph, provided they belong to separate layers corresponding to distinct unknown functions.

When REVDECODE retains only the top m candidate matches, the size of V will be $n(m+1)+2$, including one uncertain node per layer and the start and end nodes. The total number of edges in the graph will be $2(m+1)+(n-1)(m+1)^2$ to account for both the start and end nodes' edges as well as all of the edges between nodes within each layer. Therefore, the graph has a polynomial spatial complexity, with the vertices being $O(nm)$ and the edges being $O(nm^2)$.

12.2 Scores Calculation

Section 3.1.1 explains how REVDECODE encodes contextual information by assigning weights to edges in the constructed graph. The final function rankings are determined by computing the maximum-weight path through the graph. Each edge weight is a composite of four components: similarity, adjacency, confidence, and library scores. The similarity score is obtained directly from the underlying function matcher. This subsection details how the remaining three scores are calculated, along with the relevance score used to evaluate ranking quality, as introduced in Section 5.

12.2.1 Confidence Score Calculation

The confidence score calculation is derived from the Term Frequency-Inverse Document Frequency (TF-IDF) statistical measure [20], and the calculation for matching one function with another is calculated as:

$$\text{conf\_score} = \sum_{f \in F_{\text{common}}} \text{TF-IDF}(f) - \sum_{f \in F_{\text{unique}}} \text{TF-IDF}(f) \quad (7)$$

Here, F_{common} represents the set of common features between the two functions, and F_{unique} represents the set of features unique to one of the functions.

12.2.2 Adjacency Score Calculation

As described in Section 3.1.1, the adjacency score is a measure of how similar the source and destination nodes are in terms of their library name, version, optimization level, and compile unit. Algorithm 1 shows the calculation of the adjacency score.

Algorithm 1: Adjacency Score Calculation

```
Data: source_node, dest_node
Result: Value of adj_score for given nodes
if source_node.lib_name = dest_node.lib_name then
    adj_score ← adj_score + 0.7;
    if source_node.lib_ver = dest_node.lib_ver then
        adj_score ← adj_score + 0.03;
        if source_node.lib_opt_level =
          dest_node.lib_opt_level then
            adj_score ← adj_score + 0.02;
    if source_node.compile_unit =
      dest_node.compile_unit then
        adj_score ← adj_score + 0.05;
```

12.2.3 Library Score Calculation

The library score is a measure of uniqueness of the candidate's library in the context of the corpus. It is calculated as:

$$\text{lib\_score} = 1 - \frac{nf_{lib}}{nf_{corpus}} \quad (8)$$

where nf_{lib} is the total number of functions in the candidate's library and nf_{corpus} is the number of functions in the corpus.

12.2.4 Relevance Score Calculation

Following the methodology outlined in Section 5, we adopt tie-aware NDCG as the primary evaluation metric. Relevance scores are assigned to candidate functions to reflect their potential utility to a reverse engineer, based on their degree of similarity to the ground truth function. These scores range from 0 to 15. A special case arises when both the candidate and the ground truth function belong to the HAL library, due to their high code similarity; in this case, the relevance score is set to 3. Algorithm 2 shows the specific assignment of relevance scores.

Algorithm 2: Relevance Score Calculation

```
Data: Ground_Truth, candidate_function
Result: Relevance score of candidate_function
relevance_score ← 0;
if candidate_function.library_name =
  Ground_Truth.library_name then
    relevance_score ← relevance_score + 4;
    if candidate_function.function_name =
      Ground_Truth.function_name then
        relevance_score ← relevance_score + 3;
        if candi-
          date_function.library_optimization_level =
          Ground_Truth.library_optimization_level
          then
            relevance_score ← relevance_score + 1;
    if candidate_function.library_version =
      Ground_Truth.library_version then
        relevance_score ← relevance_score + 2;
else if candidate_function.library_name = HAL and
  Ground_Truth.library_name = HAL then
    relevance_score ← relevance_score + 3;
if relevance_score = 10 then
    relevance_score ← 15;
```

12.3 Embedded Libraries

Table 2 summarizes the libraries used in our Frankenbinaries construction. For each version of each library, we applied optimization levels `-O0` through `-O3` and `-Os`.

12.4 Excluded Binaries in General Purpose Evaluation Set

Table 3 lists the binaries that were excluded from the general purpose evaluation dataset due to limitations in the BSim implementation provided by Ghidra. Although this only appears in the BSim, we excluded these binaries from the evaluation set for every matcher to ensure consistency and fairness across all matchers.

Library	Description	Versions
FreeRTOS	Real-time operating system kernel for embedded devices	7.2.0, 8.0.0, 9.0.0, 10.0.0
STM32F0 HAL	Hardware abstract libraries for STM32 F0 series	1.5.0
MbedTLS	open-source TLS library	2.15.1
AWS Mqtt	Implementation of Mqtt protocol	1.1.3
STM32F1 HAL	Hardware abstraction libraries for STM32 F1 series	1.0.0, 1.8.0
STM32F4 HAL	Hardware abstraction libraries for STM32 F4 series	1.25.0
STM32L4 HAL	Hardware abstraction libraries for STM32 L4 series	1.7.0, 1.15.0
STM32F7 HAL	Hardware abstraction libraries for STM32 F7 series	1.16.0
STMUSB Library	USB driver for STM32	1.2.0
UGUI	open source graphic library for embedded systems	2.15.1
libopencm3-F1	open-source firmware library for STM32-F1	0.8.0
OpenSSL	open-source Toolkit for the TLS, DTLS and QUIC protocols	1.1.1, 3.0.8, 3.1.0
OpenCV	open-source computer vision library	3.4.15, 4.5.1, 4.7.0
Mosquitto	open-source server implementation for MQTT protocol	1.5.11, 1.6.15, 2.0.15
MbedTLS	library for cryptography and SSL/TLS protocols	2.15.1
CycloneDDS	open-source implementation of the OMG DDS specification	0.8.2, 0.9.1, 0.10.3
Aubio	a library to label music and sounds	0.4.7, 0.4.8, 0.4.9
Stunnel	open-source SSL encryption proxy	5.00, 5.50, 5.69
STM Peripheral Drivers	Peripheral drivers for STM32 L1 series	1.1.1

Table 2: Summary of Corpus Libraries

Binary	Versions
as	2.42, 2.43
ld	2.38, 2.42, 2.43
ld.bfd	2.38, 2.42, 2.43
objdump	2.42, 2.43
libcrypto	3.0.8, 3.2.1, 3.5.0

Table 3: Binaries Excluded from General Purpose Evaluation Set