IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 4, 2021, accepted July 19, 2021, date of publication July 26, 2021, date of current version August 2, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3099429

Efficient Features for Function Matching in Multi-Architecture Binary Executables

SAMI ULLAH, WENHUI JIN, AND HEEKUCK OH, (Member, IEEE)
Department of Computer Science and Engineering, Hanyang University, Ansan 15588, South Korea

Corresponding author: Heekuck Oh (hkoh@hanyang.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science and ICT under Grant NRF-2019R1A2C2003045.

ABSTRACT Binary-binary function matching problem serves as a plinth in many reverse engineering techniques such as binary diffing, malware analysis, and code plagiarism detection. In literature, function matching is performed by first extracting function features (syntactic and semantic), and later these features are used as selection criteria to formulate an approximate 1:1 correspondence between binary functions. The accuracy of the approximation is dependent on the selection of efficient features. Although substantial research has been conducted on this topic, we have explored two major drawbacks in previous research. (i) The features are optimized only for a single architecture and their matching efficiency drops for other architectures. (ii) function matching algorithms mainly focus on the structural properties of a function, which are not inherently resilient against compiler optimizations. To resolve the architecture dependency and compiler optimizations, we benefit from the intermediate representation (IR) of function assembly and propose a set of syntactic and semantic (embedding-based) features which are efficient for multi-architectures, and sensitive to compiler-based optimizations. The proposed function matching algorithm employs one-shot encoding that is flexible to small changes and uses a KNN based approach to effectively map similar functions. We have evaluated proposed features and algorithms using various binaries, which were compiled for x86 and ARM architectures; and the prototype implementation is compared with Diaphora (an industry-standard tool), and other baseline research. Our proposed prototype has achieved a matching accuracy of approx. 96%, which is higher than the compared tools and consistent against optimizations and multi-architecture binaries.

INDEX TERMS Binary diffing, efficient features, function matching, multi-architecture.

I. INTRODUCTION

An assembly function can have either a graphical representation – like its control flow graph (CFG), call graph (CG), data flow graph (DFG), etc. or a textual representation – like its assembly code, intermediate representation (IR) code, embedding vectors, etc. However, some research works [1]–[3] also formulates a new graphical representation of a function by merging the semantic or multiple structural properties into a single comprehensive graphical structure. Considering the NP nature of graph isomorphism, most of the existing function matching research [4]–[8] extract syntactic features from the structural representation and formulate an approximate solution based on extracted features. The accuracy of function matching is directly linked with the selection of efficient features that depict the approximation accuracy. As a diverse feature set can boost the accuracy of function matching, research works [6], [9], [10] extend their feature set by extracting the semantic features from the function assembly code. Research works like FOSSIL [11] adopt an alternative approach to resolve graph isomorphism. They opt to boost the efficiency of CFG matching (graph based) but these techniques are computationally expensive (NP class), not evaluated for the multi-architecture binary matching problem, and out of the scope of this research.

Function matching research is shared by malware detection, code plagiarism/clone detection, and binary diffing. Malware and code plagiarism detection research objectives are to find code similarity thus their matching algorithm even matches *partially same* functions. In contrast, the binary diffing objective is to find similarity and differences [12] between binary functions thus it first maps (*phase I*) the

The associate editor coordinating the review of this manuscript and approving it for publication was Shahzad Mumtaz.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

similar functions, later finds the small differences in functionality between the mapped functions and finally classifies (*phase II*) them into exact vs partial functions. The function matching part is common in plagiarism detection and binary diffing (*phase I*) techniques, although different objectives. Finding a higher number of *true* matches is linked with the diversity of function features thus features efficiency is critically important. Given two binaries, feature-based function matching algorithms can find the *same* functions with higher accuracy. However, function matching accuracy drops; when functions are partially modified. For partially changed functions, the accuracy of function matching could be affected due to two types of changes (i) structural changes in CFG (ii) semantic changes in assembly code. It is always risky to match two structurally changed functions as there is a probability that these functions might have real patch-based changes, optimization-based changes, or multi-architectures. To reduce the probability of false positives, most function matching algorithms first map the same matches and later map the partially changed functions. Similarly, there can be slight changes in assembly code and thus setting the threshold for sequence matching is risky; for the same reason given above. There must be some semantic aware approach [13] that can suppress the optimization introduced changes and highlight the real patch-based changes.

There are various well-known binary diffing tools such as Diaphora [10], BMAT [14], [15], and zynamics-BinDiff [5] that could be analyzed to study the impact of function features in a multi-architectures scenario. Unfortunately, BMAT and BinDiff are not open-sourced and BinDiff binaries utilize *function name symbols* thus it cannot be considered for a fair analysis. In this research, we have explored Diaphora, which is an industry-standard (open-sourced) binary diffing tool and it uses the heuristics-based selection criteria for initial function matching and utilizes the sequence matching to find the small difference in matched functions. In Diaphora, a heuristic is defined as an SQL statement formulated from a single or a few function features, which is used as selection criteria to match functions in the 1:1 mapping phase. There are 51 heuristics in Diaphora, which covers most of the proposed features in previous research works. We have empirically analyzed and studied their individual impact on the function matching accuracy. Diaphora has categorized its heuristics into three types as shown in Table 1. Its function matching algorithm selects heuristics from *HEUR_TYPE_NO_FPS* to *HEUR_TYPE_RATIO_MAX* categories in sequential manner and 1:1 map the functions. In our empirical analysis, we have found that some heuristics in *HEUR_TYPE_NO_FPS* category were over-trusted (Bytes hash, function hash, etc.) and consequently caused many false positives (especially for the functions with less than 10 instructions). Heuristics based on textual properties (opcode, strings, etc.) are only optimized for x86 architecture, and their performance drops significantly for ARM compiled binaries. Moreover, tightly stripped binaries have very little symbolic information and most of (60-70%) the function name symbols are stripped. For tightly stripped binaries, the performance of some heuristics drops significantly that were defined using function symbol-based (e.g. function strings, sequence matching) features.

TABLE 1. Heuristic types and matching criteria in diaphora.

Category	Criteria		Count
HEUR_TYPE_NO_FPS	Best	1.0	7
HEUR_TYPE_RATIO	Best	1.0	29
	Partial	$1 > x > 0.5$	
	Unreliable	< 0.5	
HEUR_TYPE_RATIO_MAX	Best	1.0	17
	Partial	$1 > x >$ threshold	
	Unreliable	$<$ threshold	

Techniques like discovRE [8], and VSkLCG [16] have proposed efficient features for searching vulnerability functions in cross-architecture scenario. Finding an exact 1:1 mapping between cross-architecture functions is a complex problem and most of the algorithms adopt a KNN based algorithm and matched functions are reported as top 1%, to top 10% with different settings of *k* parameters. The matching accuracy for the top 1% is not significant and also, these works are not evaluated for different compiler optimizations. However, DeepBinDiff [3] proposes an embedding-based graphical representation, which compares single architecture binaries but resilient to cross-optimization level binaries comparison.

In brief, there could be four possible **drawbacks** in existing function matching techniques.

- The features/heuristics might be fine-tuned only for a single architecture and not optimal for multi-architectures.
- Some features using symbolic information might not be optimal for tightly stripped binaries
- Features might not be optimal for cross-optimization levels
- In function matching algorithms, mostly heuristics are used in sequential fashion i.e. one-by-one as selection criteria. Prioritizing the best heuristic is challenging as a single heuristic can never be 100 % accurate. Consequently, the feature matching criteria is not flexible to small changes in partially changed functions and cause many false matching results.

This research aims to address the above-mentioned drawbacks in function matching for multi-architectures and propose a *general* function matching technique that is efficient towards these challenges. To cope with the first three challenges, we have critically analyzed Diaphora [10] features and proposed efficient syntactic features that maintain their efficiency for multi-architectures. Semantic features can be important in the multi-architecture scenarios and possibly boost the accuracy of function matching but they can also be a bottleneck for stripped binaries if features rely on debug information. In this research, we represent the assembly to an intermediate representation using Radare2 ESIL and extract semantic features, which are not dependent on symbolic information thus suited for stripped binaries and also expandable to multi-architectures. We propose an embedding-based

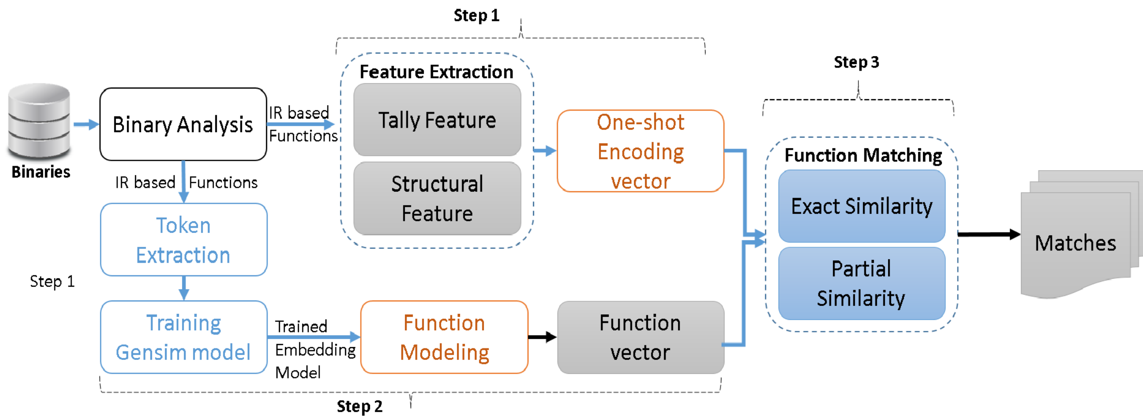


FIGURE 1. The overall workflow of the adopted methodology.

function representation that models assembly to a single vector while being sensitive to the semantic relationships among assembly tokens. This feature is not tightly dependent on the structural information, thus resilient against compiler optimizations. To cater to the fourth challenge, our function matching algorithm utilizes a one-shot encoding mechanism with distance-based selection criteria that treats all features with the same priority and it is also computationally efficient. Consequently, the distance-based selection criteria (with 10% threshold) in the function matching algorithm is flexible to partially changed function.

A. APPROACH OVERVIEW

The overall workflow of the proposed methodology is shown in Figure 1. Given a set of binaries, we initially perform the binary analysis using Radare2 and represent binary functions with their intermediate representation code that assist to resolve the multi-architecture problem. As a first *first* step, we extract the syntactic features (i.e. numeric (tally) and structural features) from each binary function and concatenate them to a single vector, which we call as *one-shot encoding* vector. As a *second* step, we filter and process the IR tokens from binary functions to generating training corpus, and train the *gensim* model that learns the embedding vector against each token in corpus. (Section III-G). The training process learns the semantic relationship among IR tokens and further helps to suppress the differences caused by different architectures and optimization levels. Once we have the embedding vectors against each token, we model each function to its equivalent vectorized representation that is used as a feature (section III-G2). *Step* 1 and *Step* 2 run in parallel and both output a feature vector, which are used as an input to function matching algorithm. As a *step* 3, we compare the function' feature vectors and find the matched functions. The complete details are in section IV.

To evaluate the proposed solution, we have implemented a prototype utility and compared it with Diaphora [10], Reveal [9], DeepBinDiff [3] and other baseline works for function matching. The evaluation parameters are the count of true positives, false positives for the matched functions, and count of unmatched functions (false negatives). Multiple experiments have been performed using XNU kernels, Coreutils, and OpenSSL binaries, which were compiled for multi-architectures. The proposed utility outperforms in most evaluation parameters, among compared tools: making it a practically useable solution for multi-architecture binary comparison.

The contributions of this paper are as follows:

- Proposed efficient features for function matching, which are efficient for binaries compiled for multi-architecture and cross-optimization levels.
- Proposed a vectorized one-shot encoding algorithm that achieves the function matching accuracy of ≈96%.
- Implementation[1] of proposed features and techniques.

II. RELATED WORK

In this section, we describe the underlying research related to function matching. Function matching has been used in various reverse engineering techniques either partially or as a full-stack approach.

A. BINARY DIFFING

Function matching is partially utilized in a binary diffing technique. A binary diffing technique first formulates a 1:1 mapping between binary functions and later classifies them into exact, partial, or no-match. Function matching is used in the 1:1 mapping phase and serves as the backbone for binary diffing techniques. Existing research focuses on finding efficient structural features that can better approximate the graph-isomorphism problem. The first technique was proposed by Flake [4], which was later extended by his colleagues Dullien and Rolles [7] and implemented in their tool zynamics-BinDiff [5]. Zynamics-BinDiff employs the feature vectors; extracted at the basic block level, to locate perfect matches and they also endeavor to reconstruct the CFG for matching against optimizations.

[1] https://github.com/sami2316/bindiff_efficient_features

Bourquin *et al.* [17] refine Dullien's work by proposing extending features and utilize the bipartite graphs in their function matching algorithm. Eschweiler *et al.* [8] extract a set of syntactical features with an aim to boost up the function analysis and computation in matching algorithms. There are some works such as BinSlayer and discovRE [8], [16], [17], which mainly focus to refine CFG that eventually boosts up the CFG isomorphism for multi-architectures but these techniques are brittle to instruction-level changes and not evaluated for cross-compilation levels.

Karamitas and Kehagias [9], [18] extends the CFG feature vector for vertices and edges that can better classify digraphs. They also employ Markov lumping technique to get a transition matrix and use it as a feature along with other features in their function matching algorithm. There are a few techniques that mainly focus on the textural features for function matching. David *et al.* [6] decomposes a function assembly to strands and computes the similarity ratio based on strands composition. Diaphora [10] is a current industry standard and an open-source binary diffing tool; which incorporates all previously researched features into a single tool. It employs different sets of features, which are computed over function CFGs and plain assembly text. Its diverse feature set can better match unique functions in two executables.

B. BINARY CODE CLONES

Nowadays, code reusability is a norm in the software development process. Many techniques have been developed to find code clones, which lay the foundation for applications like refactoring, detecting bugs, and protecting intellectual property. Finding the clones is comparatively easy [19], when the source code is available. For binaries like firmware, finding the clones at the binary level is challenging [20]. Dullien *et al.* [21] formulate a correlation technique by defining five function features that can map the code clones. Saebjornsen *et al.* [22] model the assembly instruction by extending the tree similarity framework based on clustering of characteristic vectors. They focus on assembly instruction features and preserve its structural information. Hemel *et al.* [23] define textual features, which are unique and used in their binary clone search algorithm. Their major focus was to find the code license abuses at the binary level and they have also implemented a tool called BAT. Chandramohan *et al.* [24], [25] extract the function call traces from CFG and semantic features from the assembly code of a function and propose a clone searching solution that is scalable and resilient in matching cross-architecture binaries. Alrabaee *et al.* [1], [2], [11] formulate a new graphical representation by first extracting the semantic features and then merging them into a joint data structure. Their new representation is comprehensive and effective against single architecture but not evaluated for multi-architecture and cross-optimization levels compiled binaries.

In the malware domain, Farhadi *et al.* [26], [27] model the code regions in assembly code, and their region comparison algorithm generate binary vectors, which are partitioned and hashed for comparison. Cesare *et al.* [28] defines string-based features by splitting them to n-grams. The minimum distance between feature vectors is used as a selection criterion to find the malware clone. Their features depend on function strings, which makes it inefficient for tightly stripped binaries. Xin Hu *et al.* [29] employ instruction-level features to compute graphs similarity and propose a multi-resolution indexing scheme that is scalable to large malware databases.

There are many other neural network-based techniques [3], [30]–[36], which propose a neural network-based solution to find the code similarity between binary functions. These techniques are featureless and assembly representations are learned by the neural network. Neural network-based techniques are trained in a supervised fashion, which are dependent on the labeled datasets. Labeling binary functions for comparison is a tedious job and there is no labeled dataset in the public domain. Hence, the neural network techniques cannot be effectively expandable for unseen multi-architecture assembly data and their usability is limited.

In brief, various function features approaches have been proposed in the literature but most of them are not evaluated for multi-architectures and cross-optimization levels. The textual features are mostly based on function strings, which are not available in tightly stripped binaries. Consequently, these features affect the function matching accuracy. In this research, we have proposed efficient structural features and semantic-aware textual features that are persistently efficient for multi-architectures and resilient to optimization levels.

III. FEATURES

To address the drawbacks in function matching for multi-architectures (section I), we have derived eight efficient feature vectors; by extracting the spatial, structural, and semantic features of a binary function. For each function in a binary executable, these feature vectors are extracted and the seven of them are concatenated to a single feature vector (of dimension 1×226) that we call a one-shot encoding vector. However, a function vector (of dimension 1×400) is a compact and unique feature, which is compared as a single unit in our function matching algorithm. In the following section, we explain our proposed feature vectors one by one.

A. TALLY VECTOR

In this section, we exploit the spatial properties of a binary function and propose a feature vector that preserves the spatial properties of a function. Spatial features are mostly the count of syntactic or semantic properties of a function. Most of the spatial features are already reported in the previous research [2], [9], [16]. We acknowledge their contribution and in this research, we have shorted listed the following spatial features that are a logical step towards a complete solution.

1) Edges: The number of all edges in a function's CFG.
2) Vertices: The number of all unique vertices in a function's CFG.

3) In-Degree: The number of caller functions to a function.
4) Out-Degree: The number of function calls in a function.
5) Arguments: The number of arguments in a function signature.
6) Instructions: The number of instructions in a functions assembly.
7) Cyclomatic Complexity: The number of unique paths in a CFG of function.
8) Cyclomatic Cost: The execution cost per cycle of a function.
9) Code References: The number of references from in-question function to code section. Code references are usually the function call or jump instructions.
10) Data References: The number of references from in-question function to data section. Data references are usually the pointer to strings.

We extract the above count-based features for a given function and concatenate them to a single vector that we call a tally vector. These ten features combined together can be seen as a unique signature of an in-question function. Our tally vector is represented as follows.

$$[\#E, \quad \#V, \quad \#ID, \quad \#OD, \quad \#A, \quad \#I, \quad \#CCom, \\ \times \#CCos, \quad \#CR, \quad \#DR]$$

Each function can be represented by its control flow graph ($CFG = \langle V, E\rangle$) that reflects to its structural properties. The aforementioned tally vector can filter many similar functions but at the same time, its pruning power is limited and can introduce inaccuracies in the matching process. To resolve such imperfection, we extract the structural features solely related to edges, vertices, and their relationship among them that preserve the uniqueness of a CFG. In the following sections, we explain our structural features in detail.

B. EDGE TYPE VECTOR

A CFG consists of vertices and edges that define the relationship among vertices. Following the graph theory, edges (in a function's $CFG = \langle V, E\rangle$) can be classified into four following categories: using a depth-first search (DFS) traversal.

1) Tree Edge: Consider there exists a DFS traversed topological sort of graph $CFG = \langle V, E\rangle$). An edge (u, v) satisfying the conditions that $(u, v) \in E$ and $depth(v) = depth(u) + 1$, is called as tree edge. All green edges in Figure 2 are tree edges.
2) Forward Edge: An edge (u, v) is a forward edge, if the $depth(v) > depth(u) + 1$; that is v is pointing to a descendant vertex but it is not the part of the DFS tree. An edge from vertex 1 to vertex 8 is a forward edge, as shown in Figure 2.
3) Back Edge: The edge (u, v) is back edge, if the $depth(v) < depth(u)$; that is v is pointing to an ancestor edge but not the part of DFS tree. An edge from vertex 6 to vertex 2 is a back edge.
4) Cross Edge: If an edge (u, v) connects two vertices such that there is not any ancestor or descendant relationship. In other words, edge vertices belong to two different DFS sub-trees. An edge from vertex 5 to vertex 4 is a cross edge.

We iterate through the in-question function CFG in a DFS fashion while analyzing the edge types and count each edge type. Our edge type feature vector is the count of each four edge types and it is represented as follows.

$$[\#T, \quad \#F, \quad \#B, \quad \#C] \rightarrow [7, 1, 1, 1]$$

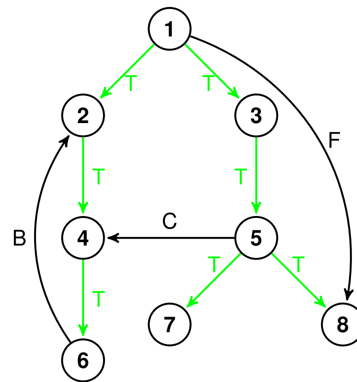


FIGURE 2. An example for CFG edges classification types.

C. VERTEX TYPE VECTOR

In a function' CFG, basic blocks are defined as vertices that can be classified based on their functionality. We follow the taxonomy of vertices followed by Tarjan [37] and Karamitas [9]. A vertex V in a function $CFG = \langle V, E\rangle$ can be classified into the following seven categories.

1) Entry: Given a function CFG, the vertices which are the root nodes in a function CFG or execution entry points are *entry* vertices. In a CFG, the basic blocks containing function prologue are entry vertices. There can be single or multiple entry vertices.
2) Exit: The vertices which are the exit nodes in a function CFG or the basic blocks containing the epilogue of a function assembly. Mostly these basic blocks end with RET instructions. There can be multiple exit vertices in a function CFG.
3) Traps: The vertices which have a single edge looping into itself but do not have any descendent vertices or outgoing edges to other vertices. Traps vertices can mostly be the last vertex in CFG paths.
4) Self-Loops: The vertices which have a single edge looping into itself but also may have descendent vertices or outgoing edges to other vertices. Traps are also self-loops but the reverse is not true.
5) Loop Heads: The vertices which are the starting point of a loop.
6) Loop Tails: The vertices which are the ending point of a loop.

TABLE 2. Possibilities of basic block in-degree and out-degree combinations.

		outdegree			
		0	1	2	≥ 3
indegree	0	D_{00}	D_{01}	D_{02}	D_{03}
	1	D_{10}	D_{11}	D_{12}	D_{13}
	2	D_{20}	D_{21}	D_{22}	D_{23}
	≥ 3	D_{30}	D_{31}	D_{32}	D_{33}

7) Normal: All vertices in a function CFG including those that do not fall in any of the above categories.

We iterate through in-question function CFG in DFS fashion and count the each above defined vertex types. Our vertices type feature vector is the count of each type and represented as follows:

$$[\#N, \quad \#En, \quad \#Ex, \quad \#T, \quad \#SL, \quad \#LH, \quad \#LT]$$

Although the above features play an important role for the feature comparison, they are not sufficient as they do not capture the topological relationship in a CFG. To capture the topology of CFG, we have proposed the following two simple but reliable features.

D. VERTEX DEGREE VECTOR

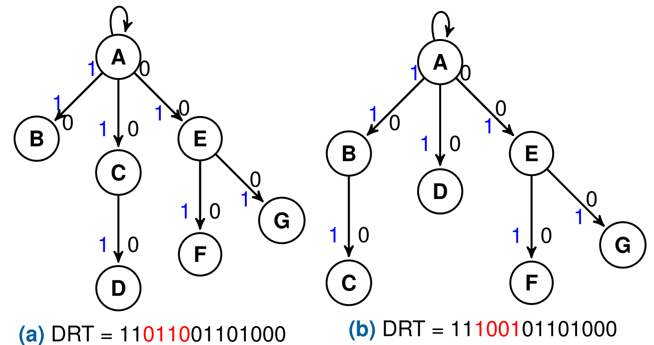
A CFG can also be represented as a set of basic blocks (vertices) and connections (edges) among them. A basic block can have a single or multiple connections coming in (i.e. vertex parents) or connections going out (i.e. vertex children) that define the in-degree and out-degree of a basic block [38]. In this research, we have empirically analyzed the Apple kernel binaries and noticed that the majority (96%) of in-degree and out-degree values range from 0 to 3. To abstract the basic blocks in-degree and out-degree relationship, we have considered only the range from 0-3 that result into 16 possible combinations. To incorporate more than 3 connections, we abstract $\geq$3 as 3; as shown in Table 2.

Our vertex degree vector is the count of each 16 combinations mentioned in 2. We iterate through the CFG in DFS fashion and update the following degree vector.

$$[D_{00}, \quad \#D_{00}, \quad \#D_{01}, \quad \#D_{02}, \quad \#D_{03}, \quad \#D_{10}, \\ \times \#D_{11}, \quad \#D_{12}, \quad \ldots \#D_{33}]$$

E. DIGRAPH DOMINANCE RELATIONSHIP (DDR)

The vertex type, edge type and vertex degree feature do not give insight into the actual layout of a CFG. Consider the case shown in Figure 3, both digraph have same vertices classification vector [7, 1, 4, 0, 1, 0, 0], edges type vector [6, 0, 0, 0] and vertices degree vector. Merely using classification based features are not feasible to discriminate the cases like shown in Figure 3. It can be seen that Figure 3 digraphs have different layouts and differ in their dominance relationship. To cope with such cases, we first build the Dominator Relationship Tree (DRT) against each function CFG, by visiting the CFG in DFS fashion and then generate a binary string by using a simple rule that when we enter a vertex, we add 1 to the binary string and add 0 when we leave the vertex. As shown in Figure 3, DRT based binary strings for both digraphs differ by at least four points and thus such cases can be uniquely represented.



(a) DRT = 11011001101000 (b) DRT = 11100101101000

FIGURE 3. Immediate dominator tree relationship.

Our previous features are in vector space but the DRT based binary string cannot be directly fitted in our model due to the following challenges.

- Each function has a different length CFG. Therefore, the outcome of DRT based binary string would be of different length. In the function matching algorithm, we need to compare a single function to all other unmatched functions that require a feature vector that must be of fixed length. If we take the longest binary string length to generate vectors and perform zero padding for smaller size binary strings, the outcome would be sparse vectors.
- Due to compiler optimizations (under same optimization level), the basic blocks might be repositioned in a CFG. The layout representation feature must consider such changes but simple sequence based binary string comparison either cannot discriminate such changes or produce false positives.

We have analyzed the real world binaries (XNU kernel binaries) and found that majority of (98%) the function DRT based binary string length lies in the 0-600 range. The histogram of DRT strings length is shown in Figure 4.

In order to deal with the varying length of DRT binary strings, we have considered the 768 (selected after empirical evaluation) as its representation value against all functions. We transform the DRT binary strings to an equivalent vectorized form and zero pad the vectors with length less than 768. As most of the functions lie in 0-150 range, so many functions DRT vectors will be sparse. To cope with the sparsity challenge, we have leveraged from the locality sensitive hashing research [38], [39] that preserve the semantic relationship as opposed to the cryptographic hashing schemes and proposed the following two hashing based vectors that solve the above challenges.

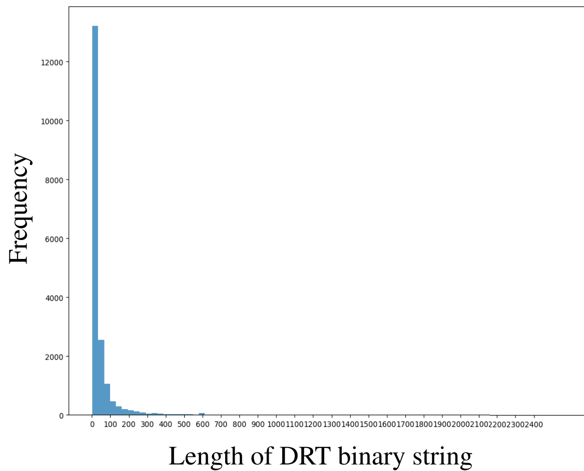


FIGURE 4. Histogram of DRT based binary strings length.

1) PIECEWISE HASHING

DRT vectors are computed by visiting a CFG in DFS fashion (as described in section III-E) thus preserving the layout of a CFG. Our DRT vector consists of 768 elements and its sparse for most of the functions in a binary. To reduce the dimensionality and impact of sparsity, we compute the piecewise hashing of the DRT vector and output is a reduced dimensionality vector. Our DRT vector is a binary vector (i.e. 1 or 0 value), so we adopt a simple approach by representing 32 bits into an equivalent *float32* format and sequentially iterate over the DRT vector to generate the reduced DRT vector of size 32. Our piece-wise hashing algorithm is given in Algorithm 1.

Algorithm 1 Piece-Wise Hashing Algorithm

Input: DRT binary string
Output: hash_vector
1: bit_vect = *format*(DRT binary string, "0<768b")
2: **for** $i = 0$ to 768 **do**
3: elem = int(bit_vect[i]:i+32,2)
4: hash_vector.append(elem)
5: **end for**
6: **return** hash_vector

2) PROJECTION BASED HASHING

Piecewise hashing preserves the CFG layout but does not consider the randomness against basic blocks repositioning. To introduce randomness, we project the DRT vector on k linearly independent planes [40] and compute a k-bit signature in vector form. The value k is much less than the DRT vector length (768), so the output k-bit signature vector is an approximation of the DRT vector. As the projection planes remain the same over all functions in each binary, so the two similar functions will have exactly similar k-bit signature vectors. In this research, we have experimented with different k values but $k = 128$ gives the optimal accuracy. As k is not large, we generate k random permutation vector (P_k) of length 768 and a projection is dot product between in-question function DRT vector and permutation vector P_k. So the k-bit signature vector is the output of dot products of all P_k with the DRT vector. Detailed algorithm is given in Algorithm 2.

Algorithm 2 Project Based Hashing Algorithm

Input: DRT binary string
Output: Sig_k vector
1: hash_size = 128
2: input_len = 768
3: uniform_planes = random_matrix(hash_size, input_len)
4: bit_vect = *format*(DRT binary string, "0<768b")
5: **for** $i = 0$ to $hash_size$ **do**
6: elem = int(dot(uniform_planes$_i$, bit_vect),2)
7: Sig_k vector.append(elem)
8: **end for**
9: **return** Sig_k vector

In brief, digraph dominance relationship preserves the layout of a CFG and it helps to differentiate between Figure 3 shown digraphs. DRT based binary string preserves the CFG layout thus our vectorized hashing schemes also respect the CFG layout and provide syntactic-aware hash signatures.

F. OPCODE VECTOR

A function can have both structural (CFG) and textual (assembly or IR) representations thus an efficient feature set must include features extracted from both representations. The previously discussed features are derived from structural properties of a function. When a function is compiled for different architectures, these structural properties do not vary; as these are not dependent on instruction sets. In contrast, a binary function compiled for different architectures utilize different instruction sets, which result in different assembly codes. A textual feature derived from assembly code can only be optimized for a single architecture. Consequently, the same textual feature extracted for different architecture assembly code will be different thus introduce errors in function matching algorithms. Previous research [18] has defined instruction and function string histograms as textual features, however both of them cannot be generalized to all architectures. In this research, we have used Radare [41] tool generated evaluable strings intermediate language (ESIL) and extracted the textual features that resolve the multi-architecture issues. ESIL has defined its own instruction set, which is the same for all architectures. To preserve the textual properties of a function, we have proposed the opcode vector that can be described as a histogram of each possible opcode types. The possible types of op-code against all architectures are described in Appendix B-A. There are a total of 48 op-code types against all architectures and our analysis on XNU kernel binaries (x86 and ARM64) shows that the opcodes in range 37-48 (see Appendix B-A) occur very rarely. In this research, we have

considered 0-36 opcode types and generated a feature vector that counts the frequency of each opcode type against each function, see the Algorithm 3.

Algorithm 3 Algorithm to Computing Op-Code Vector

Input: Assembly
Output: opcode_vector
1: opcode_vector = [0] * 37
2: **for** $i = 0$ to $len(instructions_{CFG})$ **do**
3: inst_type = get_inst_type($instructions_{CFG}[i]$)
4: opcode_vector[inst_type] += 1
5: **end for**
6: **return** opcode_vector

G. IR EMBEDDING REPRESENTATION

Opcode vector is derived from the operator token of an ESIL statement, whereas the operands are not utilized. To capture the diverse nature of textual representation, there is a need to define another feature vector. Furthermore, we have observed the redundancy in the functionality of small functions with a few basic blocks i.e. they have similar structural features even the number of instructions and opcode vectors. Merely relying on the structural features, such functions cannot be accurately matched as they are almost the same. Such functions slightly differ in their textual properties such as the caller function names, register names, etc. There is a need to better preserve the textual properties of a function, which remain the same for different architectures, stripped and unstripped binaries. Function strings based features are not optimal against stripped binaries and not expandable to multi-arch binaries. In this research, we have proposed an embedding-based feature that is based on an intermediate representation of assembly code and respects the semantic properties of a function. The proposed semantic aware feature can effectively match above mentioned cases and they are fine-tuned for multi-architectures. There are existing assembly embedding models [42], [43] that model an assembly function but the proposed model is simple and practical. We incorporate the *word2vec* [44] based embedding model [45] that respects the semantic relationship and it can effectively match two similar functions.

word2vec has embarked its significance in different data analysis tasks. In this research, we train the *gensim* implemented word2vec model using the IR tokens extracted from various kernel binaries. Context plays a key role in learning the semantic relationship among tokens. In this research, we consider an IR statement as a sentence, and each token context is defined relatively within the given IR statement. A disassembled function contains many offset constants and address tokens, which introduce randomness. From the function representation perspective, such addresses cause noise, and also the address space is enormously large as compared to the IR token set. Consequently, address-based tokens affect the learning accuracy of IR tokens. For these reasons, there is a need to filter the addresses and constants. In this research, we replace addresses with constant *'XXXX'* string and numeric constants with *'OFFSET'* string. Given the IR tokens, we train the word2vec model that learns the embedding vectors against each token. Once the model is trained, token embedding vectors are used in our function representation model that is explained in the following section.

1) IR STATEMENT EMBEDDING MODEL

Towards the multi-arch function matching features, we need to represent assembly code to an IR language. In this research, we have utilized Radare2 ESIL as an IR language and the reason for this choice is explained in Appendix B. An IR statement has a fixed syntax i.e. two or more operand terms followed by an opcode (operation) term. Most of the IR statements have the form given in the Figure 5.

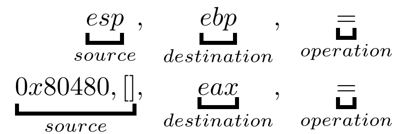


FIGURE 5. An IR statement representation.

In this research, we split an IR statement by comma (,) and categorize the tokens into *operation* vs *non-operation* terms. An operation term is always a single term while non-operation terms can be two or more. Using the pre-trained *word2vec* model, we first transform each non-operation token to their embedding vectors and aggregate them to a single summation vector. Later, the summation vector is concatenated with the operation term embedding vector and the corresponding vector is called as IR-statement vector.

2) FUNCTION MODELING

Each function in a binary can have a different number of instructions depending on its functionality. From the function matching perspective, we need to represent a function in a fixed vectorized representation that can preserve its uniqueness and relationship among other functions. Using the IR statement embedding model, we have proposed the following two representations that are used alternatively in the function matching algorithm.

- *Aggregation Vector:* Given a function F_i with n instructions, we first compute an IR-statement vector IR_n against each instruction and later aggregate all n IR-statement vectors to a single vector. For the aggregation, either sum or average can be used but we have empirically analyzed that summation achieve higher accuracy. This representation is efficient in matching the same functions.
- *Fixed Matrix:* In large binaries, there are a few functions (in different classes or namespaces) which share the same functionality and their IR representation is slightly different. Consequently, the aggregation vector is not efficient against such cases. To cater such unique cases, we have proposed a matrix representation of a function,

where each row is an IR-statement vector. For the computational efficiency, two compared matrices must have the same dimension. For the sake of same dimensionality, we select the dimension of the larger function and zero pad the other function that has less number of instructions. This matrix based representation is efficient in detecting the small instruction level differences and matching the partially same functions.

This section concludes our discussion on proposed efficient features. The proposed feature set covers all aspects of a binary function, considering a multi-architecture scenario. All seven features are in vectorized form and preserve the unique properties of the in-question function. In the following sections, we describe the methodology to use proposed features effectively and an evaluative comparison is given to prove their effectiveness under complex scenarios.

IV. PROPOSED FUNCTION MATCHING

Once features are extracted against all functions in target binaries then these features are used in the function matching algorithm to 1:1 map functions in both binaries. Previous research [9], [10] has defined heuristics based on a single or multiple (<5) function features. These heuristics are used as a selection criteria to 1:1 map the same (or similar) functions in both binaries. Their matching algorithm iterates through each heuristic (one at a time) and if two functions satisfy a heuristic criteria then they are 1:1 mapped. After all the functions are processed using all heuristics, the 1:1 mapped functions are categorized as *matched* and all remaining functions are categorized as *unmatched*.

As discussed in section I, a single heuristics can never be optimal. To cater single feature inaccuracy, we have proposed vectorized features which are concatenated to a single feature vector and then that single vector is used in a matching algorithm. We adopt a two-fold strategy i.e. (i) all structural features are concatenated to a single vector (dimension of 1×226) and compared with all other functions' structural features in one-shot fashion. (ii) Textual features (function vectors of dimension 1×400) are compared with all other functions' textural features in one-shot fashion. Proposed function matching algorithms utilize both resultant vectors and adopt an exact match first (EMF) strategy to find the same functions. To incorporate similar functions (not 100% same), we employ a flexible distance based selection criteria and match the functions which are at-least 90% similar. Consequently, proposed function matching algorithm efficiency does not diminish due to a single feature inaccuracy and it is flexible to partial changes thus resolving the third and fourth challenges. The complete details of our matching algorithm are given in the following sections.

A. FEATURES ENCODING

Some spatial or structural features like tally vector might not be optimal in cross-optimization scenario as optimization affect the CFG but they are important feature towards a general solution. In contrast to the comparison of selective features in an iterative fashion, we have proposed a one-shot encoding scheme that suppresses the individual impact of a features thus boosting performance against optimizations. Given a set of feature vectors of a binary function, we first transform them to corresponding *numpy* arrays and later concatenate them to a single representation vector. The same procedure is repeated for all functions in both binaries under comparison. The resultant representation vectors are further used in proposed function matching algorithms.

The proposed function matching algorithm uses a two-fold strategy. Therefore, there are two representation vectors against each function. First representation vector is formulated by using a feature set containing features A-F. The vector dimensions are 1×226 and shown in Figure 6.

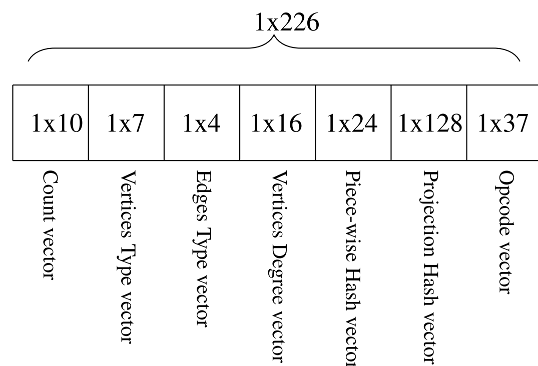


FIGURE 6. First encoding vector.

The order of the arrangements of feature vectors does not affect the matching algorithm accuracy as far as we keep the same order for all functions in binaries. In this research, the choice of arrangement of features in the first representation vector is given in Figure 6.

Second representation vector is formulated using only a single feature i.e. function vector (Section G). Its dimensions are the same like function vector 1×400. In brief, the first representation vector is a collection of all spatial and structural features and the second representation vector is a collection of textual features. The strategy to use these representation vectors in a function matching algorithm is given in the following section.

B. REPRESENTATION VECTOR MATCHING

Given two binaries A_{bin} and A'_{bin}, where A'_{bin} is the patched version of A_{bin}. Each binary consists of k and k' number of functions; which are already disassembled by a disassembler like IDA or Radare2. Suppose that we have already extracted the features against each function in both binaries and generated their representation vectors. Given a representation vector of any function in A_{bin}, algorithm given in Figure 7 is used to select the best matched function in A'_{bin}. We have applied the same algorithm for both representation vectors, separately. However, the distance functions and the threshold values are different. The details of each module in Figure 7 are as follows.

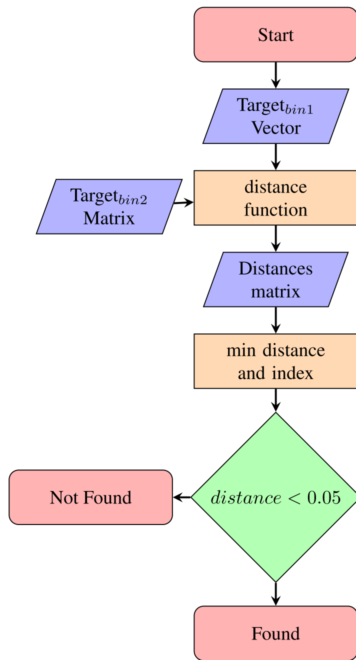


FIGURE 7. Vectorized feature matching strategy.

- **Target$_{bin1}$ Vector**: In a function matching algorithm, all functions of A_{bin} are compared one-by-one with all unmatched functions of A'_{bin}. A representation vector of in-question function from A_{bin} that will be compared with all representation vectors of A'_{bin} is defined as a Target$_{bin1}$ Vector.
- **Target$_{bin2}$ Matrix**: As our proposed algorithm is in vector space, so to improve the comparison computation, we have accumulated all representation vectors (against all functions) in A'_{bin} into a single matrix. Each row in the matrix is a representation vector against a single function from A'_{bin}. We have defined that matrix as Target$_{bin2}$ Matrix.
- **Distance Function**: As discussed in section IV-A, we have generated two representation vectors. In the first representation vector each element represents a unique feature. Two same functions must have the same unique representation vectors but in case of partially changed functions (after some patch) the features will be partially different. Changes in partially changed functions are usually small and are differentiable from the two completely dissimilar functions. In large binaries with $>20K$ functions, where we have to compare one function against $>20k$ functions, the threshold window is critically small. With empirical analysis and trying different distance functions and configurations, we have found that hamming distance with at-least 90% similarity gives the best results. We have used normalized hamming distance and the distance threshold is 0.1 (90% similarity).
  In the second representation vector, each function is represented by an embedding vector, which preserves the semantic relationship. In this case, we employ the cosine distance to measure the similarity between two functions. If two functions are the same then the cosine distance is zero. To match the partially changed functions, we employ a threshold of 0.05 (95% similarity) and if the distance is more than threshold, we consider them dissimilar functions.
- **Min Distance**: The output from the distance function is a distance matrix where each row$_k$ represents the distance of Target$_{bin2}$ vector with the row$'_k$ of Target$_{bin2}$ Matrix. To formulate the 1:1 mapping, we select the row$_k$ (of distance matrix) with minimum distance. If the minimum distance is less than threshold, we consider it a match and map the index k' of Target$_{bin2}$ Matrix with the index k of Target$_{bin1}$ Vector.

This section explains the procedure to efficiently match representation vectors. With this, we are all set to explain our main function matching algorithm, which is explained in the following section.

C. FUNCTION MATCHING ALGORITHM

Function matching is a process of finding a 1:1 mapping between the functions of A_{bin} and A'_{bin} binaries. In the proposed function matching algorithm (FMA), we make use of feature representation vectors and section IV-B algorithm to compare binary functions and find a 1:1 mapping between them. The detailed algorithm is given in Algorithm 4.

In Algorithm 4, there are two important base functions that lay the foundation in defining the FMA strategy i.e. *STRUCTURAL_MATCHING()* and *TEXTUAL_MATCHING()*. These functions are defined with section IV-B algorithm; using the first and second representation vectors, respectively. These functions compare the representation vectors and 1:1 map the indexes with the minimum distance criteria.

The exact matches are defined as the mapped functions, which have zero distance between their representation vectors. In the proposed FMA, we first find the exact matches as there is less probability that two dissimilar functions will have zero distance between them. Our empirical analysis shows that there are a few (0.01%) false positives in mapping the exact matches as compared to mapping partial matches. *FIND_STRUCT_EXACT_MATCHES()* and *FIND_TEXT_EXACT_MATCHES()* functions define the strategy to find the exact matches based on the structural and textual representation vectors, respectively. *FIND_STRUCT_EXACT_MATCHES()* focus on finding the structurally similar functions and for cross verification, we ensure the zero cosine distance between function vectors (textual feature) of mapped functions. In contrast, *FIND_TEXT_EXACT_MATCHES()* focus on finding the textually similar functions and map the function when the cosine distance is less than equal to 0.1. To secure the accuracy in mapping the exact functions, we employ the count of minimum distance entries in the distances matrix. When the count is one then we consider it an exact match and map

Algorithm 4 Function Matching Algorithm (FMA)

```
Input: A_bin, A'_bin feature databases (db1, db2)
Output: list of matched and unmatched
 1: function FIND_STRUCT_EXACT_MATCHES(db1, db2)
 2:     for index1 = len(db1) to 0 do                 ▷ Reverse Loop
 3:         index2, dist1 = structural_matching( db1, db2)
 4:         if count(min(dist1)) == 1 then
 5:             dist2 = function_vec_matching(index1, index2)
 6:             if dist2 == 0 then
 7:                 matched.append(index2)
 8:             end if
 9:         end if
10:     end for
11: return matched
12: end function
13: function FIND_TEXT_EXACT_MATCHES(db1, db2)
14:     for index1 = len(db1) to 0 do                 ▷ Reverse Loop
15:         index2, dist1 = textual_matching( db1, db2)
16:         min_d = min(dist1)
17:         if count(min_d) == 1 & min_d <= 0.1 then
18:             matched.append(index2)
19:         end if
20:     end for
21: return matched
22: end function
23: function FIND_PARTIAL_MATCHES(db1, db2)
24:     for index1 = len(db1) to 0 do                 ▷ Reverse Loop
25:         index2a, dist1 = textual_matching( db1, db2)
26:         top_d, top_i = KNN(dist1,k)               ▷ Top k matches
27:         for i ∈ top_i do
28:             index2b,dist2 = function_vec_matching( index1, i)
29:         end for
30:         matched.append(index2b[min(dist2)])
31:     end for
32: end function
33:
34: matched += FIND_STRUCT_EXACT_MATCHES (db1, db2)
35: matched += FIND_TEXT_EXACT_MATCHES (db1, db2)
36: matched += FIND_PARTIAL_MATCHES(db1, db2)
37: Unmatched = remain unmatched functions
```

the indexes of in-question function and minimum distance function.

Once all exact matches are found, then we find the partial matches using *FIND_PARTIAL_MATCHES()* that follow a similar strategy but it is thresholded at 90% similarity to 1:1 map functions. It is the case, where there are more than one functions with same minimum distance, we use the KNN and select top K matches and compute the cosine distance of the in-question function with each top K function' textual representation vector. The function with the minimum cosine distance is considered a match and it index is mapped with the in-question function index.

For large binaries, comparing a function representation vector with all functions representation vectors is computationally expensive. Once a function is mapped, we removed it from the dataset and process the remaining functions. Practically, this strategy boost the computational efficiency of function matching algorithm.

TABLE 3. Binary dataset for training, testing, and evaluation.

Type	Version	Functions count: Total Detected	Distinct	Added/ Deleted
Apple kernels	XNU-4570-31.1	18,167	17,802	–
	XNU-4570-41.2	18,171	17,806	8/1
	XNU-4570-51.1	18,230	17,864	152/172
	XNU-4570-61.1	18,235	17,869	6/1
	XNU-4570-71.1	18,241	17,873	13/4
Linux kernels	Linux-4.4.235	43041	41531	17/2
	Linux-4.4.236	42973	41547	
	Linux-5.8.8	58600	57433	7/2
	Linux-5.8.9	58712	57444	

V. EVALUATION

The proof of concept implementation is coded in python and consists of two major utilities. (i) Feature extraction utility: it uses the *r2pipe* wrapper module (for *Radare2*) to first analyze a binary and later extract the feature vectors against all functions. Extracted features are saved in a database for later comparison. (ii) Feature comparison utility: it implements the FMA, which compares the function features, extracted by the feature extraction utility. The FMA output four files in cPickle format that are the exact matches, partial matches, unmatched in primary and secondary binary files, respectively.

The experiments were run on a Ubuntu machine with a core i7, 64Gb RAM, and Nvidia 2080ti GPU. For the evaluation, we have used kernel binaries that were large and diverse enough to serve the evaluation objectives. The details of each compared kernel binaries are given in Table 3. Furthermore, the *highlighted* XNU kernel versions (in Table 3) are the ones used to generate the dataset for the training *word2vec* module. In the following, we explain the adopted procedure to obtain the kernel binaries.

A. KERNEL BINARIES EXTRACTION

In this section, we explain the procedure to collect binaries for evaluation. ARM, x86, and MIPS have been widely deployed architectures and used for evaluation in previous research. However, in this research, we have considered only stripped and unstripped versions of x86 and ARM. At the time of writing, *Radare2* (binary analysis tool) does not fully

support MIPS architecture so we have skipped the evaluation for MIPS, and evaluation is limited to ARM and x86 (Linux, Darwin) compiled binaries.

1) x86 ARCHITECTURE

For the evaluation of x86 binaries, we have utilized *stripped* and *unstripped* kernel binaries. To keep the uniformity of the same kernel versions for each x86 stripped, x86 unstripped, and ARM stripped binaries, we have selected the APPLE XNU kernel that empowers both macOS and iOS. In macOS, the officially distributed kernel binaries are stripped binaries. To obtain official binaries, we have installed different MacOS updates and directly collect the kernel binaries from the root directory. To collect the unstripped binaries, we manually compile the XNU source code[2] using the build script. The compilation output is unstripped and stripped binaries but we have used only the unstripped binary for evaluation. The detailed procedure for the compilation is described in the build script that is open-sourced here.[3]

2) ARM ARCHITECTURE

For the evaluation of ARM binaries, we use only the stripped binaries. Due to the proprietary issues, the open-source XNU includes the tempered iOS code thus source code compiled binaries cannot be used for a fair ARM architecture evaluation. In macOS, kernel binaries can be directly collected but in iOS, the root directory is locked to avoid jailbreaking. There is no direct way to collect kernel binary from iOS (ARM), so we adopt the following indirect procedure and extract iOS kernel binaries.

1) Download the complete iOS binary file from https://ispw.me. Replace the *.ipsw* extension with the *.zip* and unzip the file (iphone<version>).
2) To collect the kernel binary from unzipped file, it is further processed using *jtools*[4] $ jtool2 -dec iphone<version>
3) It will write the kernel to */tmp* folder that can be directly copied and used for analysis. The resultant file is directly analyzable in IDA or Radare2.

Following the above procedure, we collect ARM compiled kernel binaries and use them for our analysis.

B. COMPARISON FOR THE INDIVIDUAL IMPACT OF REPRESENTATION VECTORS

Previous research on function matching either immensely focuses on *syntactic* or *semantic* representations of a function. However, for the proposed function matching features, we keep a balance and potentially benefit from both syntactic and semantic features. To emphasize the importance of each representation vector, and analyze their individual impact; in this section, we have evaluated the impact of (i) structural vs textual features, and (ii) proposed function modeling vs Asm2vec [43].

1) SYNTACTIC VS SEMANTIC FEATURES

In this section, we have compared XNU-4570-31.1 ⊖ XNU-4570-41.2 to evaluate the individual impact of *syntactic* and *semantic* features. For analysis, the selection of kernel binaries is random and comparison results are plotted in Figure 8. However, the evaluation results are consistent for other binaries as well. In our experiments, we consider all proposed spatial and structural features as a single syntactic representation vector (first) and an embedding-based function vector as a semantic representation vector (second). The evaluation parameters are the count of true positive (TP), false positives (FP), and unmatched (UN) or false negative (FN) functions.

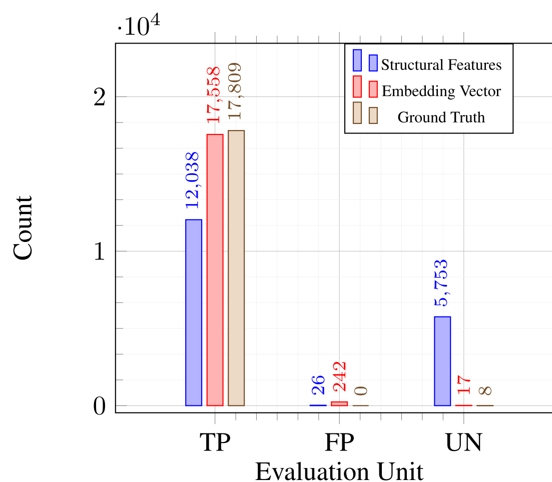


FIGURE 8. **Comparison of *structural* representation vector vs *textual* representation vector.**

As shown in Figure 8, semantic representation vector can match more number of functions as compared to the syntactic representation vector. Structural features preserve the distinct properties of a function and even small changes in function' CFG are amplified. Consequently, the FMA can match only exact matches with precision, which are strictly the same. In contrast, the embedding model learns to find the semantic similarity/differences between assembly tokens and aggregation in Algorithm 3 further suppresses the minor difference. As a result, FMA can match more functions that have semantic similarity but this suppressing effect ignores small changes and causes more false positives. Overall, the embedding-based features outperform structural features in FMA settings. In brief, it can be concluded that the embedding-based representation is a promising solution for matching functions with accuracy and comparatively better than structural features. However, both types of features are equally important and their combined effect is studied in Section V-C.

2) PROPOSED EMBEDDING MODEL VS ASM2VEC

In this section, we have compared the proposed embedding model with Asm2Vec [43], which is a well-known assembly

[2] https://opensource.apple.com/source/xnu/
[3] https://gist.github.com/sami2316/467a8dc82493e8b748924eda2ab23c84
[4] http://newosxbook.com/tools/jtool2.tgz

to vector representation model. For comparison, we have used XNU-4570-31.1 ⊖ XNU-4570-41.2 kernel binaries (relatively unseen dataset) and evaluation parameters are the count of *true* positives, *false* positives and unmatched (FN) functions. In our experiments, we have utilized the *unofficial* implementation of Asm2Vec, open-sourced on Githhub.[5] The Asm2Vec model is not meant to be a function matching algorithm. To study its impact in a function matching scenario, we have implemented[6] the prediction model using Asm2vec trained weights and merged it in our FMA. The evaluation is conducted for Asm2Vec+FMA and proposed model+FMA and the comparison results are plotted in Figure 9.

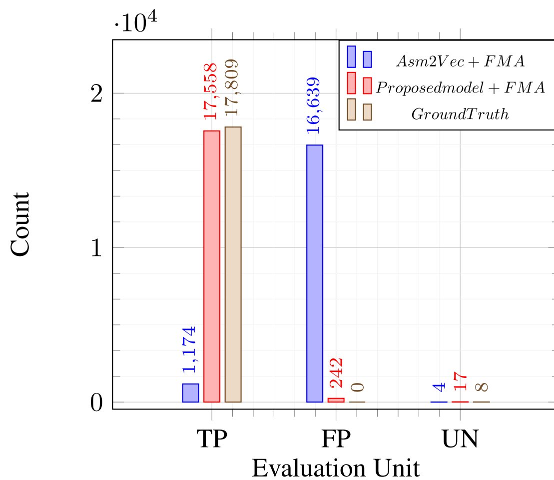


FIGURE 9. Comparison of proposed embedding model with Asm2Vec.

To train the embedding models (Asm2Vec and proposed), we have used Table 3 highlighted kernel versions. However, the prediction model in Asm2vec *unofficial* implementation always retrains the network, whenever new vocabulary words are found, but the proposed model is only trained once and never retrained or trained with this section evaluation dataset. As shown in Figure 9, Asm2Vec performance is worst compared to the proposed embedding model. There are more than 90% false positives in the matched results. The reason for this inaccuracy for Asm2vec might be linked with a fewer training datasets or its incompatibility with the function matching scenario. The improved efficiency of the proposed model is linked with the two factors (i) it uses the ESIL as IR that filter the assembly noise and splits the IR statements into tokens which are finite in nature and can be efficiently learned with a fewer dataset. (ii) Algorithm 3 based function vector computation respects the structural properties of functions and hence the model output vector captures diverse properties and efficient. In brief, it can be concluded that an embedding model which suppresses the assembly noise and preserves the structural properties of functions, can be effectively used in FMA. This effect is further studied in section V-H evaluation results.

[5]https://github.com/oalieno/asm2vec-pytorch.git
[6]https://github.com/sami2316/asm2vec-pytorch

3) EVALUATION FOR UNSEEN DATASET

In this section, we have evaluated the accuracy of the proposed embedding model for the completely unseen dataset. The evaluation parameters are the count of true positives, false positives, and unmatched functions. For comparison, we have selected four Linux kernel versions (compiled for x86 architecture) and compared them with their immediate kernel versions, which end up in two binary pairs. The ground truth for these binaries is reported in Table 3 and the comparison results with *Asm2vec* are shown in Figure 10.

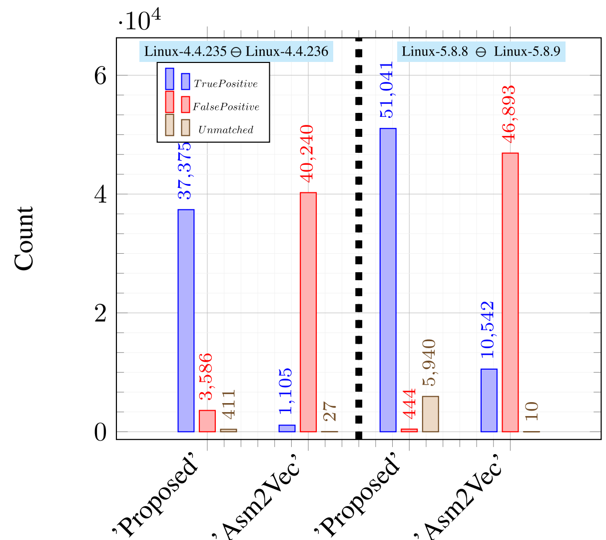


FIGURE 10. Comparison for the unseen dataset.

As shown in Figure 10, Asm2vec performance is worst and also consistent with the previous section results. However, there is a slight drop (around 8-10%) in the efficiency for the proposed model, when evaluated on a completely unseen dataset. ESIL mnemonic and opcodes are finite in nature (learnable with a small dataset) but there exist function strings, which cannot be completely learned with a small dataset. In our function vector generation model, an unseen token is replaced with a *zero* vector, which seeds in a drop of some useful information for a function and consequently causes mismatches and false positives.

For the proposed model, training is not a big effort and a model can be retrained for the unseen tokens. However, this inaccuracy for the unseen dataset can be also be fine-tuned [2], [10] with the parallel use of structural features in FMA. The proposed FMA potentially utilizes both structural and function embedding vectors to match the binary functions and thus boosts the accuracy.

C. COMPARISON WITH THE BASELINE TOOLS

The function matching algorithm (FMA) can classify the binary functions being matched or unmatched (usually the cases of newly added or deleted functions). In the matched functions, if the FMA rightly maps the two functions then we call it a True Positive (TP) and for wrong mapping, we call it a False positive (FP). As ground truth, we compare the names

of the mapped functions to describe a TP or an FP case. In the unmatched functions, we report the functions which do exist in both binaries but the FMA cannot match them and wrongly report them as unmatched. In our evaluation, the performance is measured in terms of the count of TP, FP, and unmatched functions that together depict the FMA accuracy.

$$SELECT\ COUNT(*)\ FROM\ results\ WHERE\ type = \text{``best''}; \tag{1}$$

$$SELECT\ COUNT(*)\ FROM\ results\ WHERE\ type = \text{``best''}\ AND\ (name{=}name2\ OR\ name\ LIKE\ \text{``sub_\%\%''}\ AND\ name2\ LIKE\ \text{``sub_\%\%''}); \tag{2}$$

$$SELECT\ COUNT(*)\ FROM\ unmatched\ WHERE\ type = \text{``primary''}; \tag{3}$$

To evaluate and compare the performance of our proposed function matching technique with the baselines, we have compared it with Diaphora and Reveal [9]. Diaphora is developed by Joxean Koret and it is open-sourced at Github.[7] To compare the kernel binaries, we have used its default configurations and disabled the *same name* heuristic to truly evaluate the impact of its features. Diaphora saves its output in an SQLite format. We have used the following SQL statements in (1)-(3) to find the total matches by Diaphora, count of FPs, and count of unmatched functions, respectively.

Reveal is not open-sourced yet but we are thankful to Chariton Karamitas for sharing their code with us. We run Reveal with its factory settings on the same hardware. Its output is three separate files in cPickle format and we use similar criteria defined for Diaphora to find the total matches, count of FPs, and unmatched functions. Once the matching is done, we used the same name as ground truth to evaluate the accuracy of matched functions. Equations (1)-(3) is used as a benchmark and we have applied the same criteria for the proposed tool and analyzed the proposed FMA output.

The proposed tool uses Radare2 for binary analysis, whereas Diaphora and Reveal both uses IDA for binary analysis. Both binary analysis tools have their pros and cons, details are in appendix , but the number of detected functions for kernel binaries is differently reported by both tools. Radare2 detects more functions for x86 architecture but IDA detects more functions for the ARM architecture. The reason for this difference is out of the scope of this research. For evaluation fairness, we have computed the relative percentage for the evaluation parameters.

In the following section, we have evaluated the performance for unstripped and stripped binaries compiled for x86_64 architecture, and stripped ARM binaries. Due to the limitation of Radare2, we have selected only the x86_64, and ARM architectures; for this section evaluation.

[7]https://github.com/joxeankoret/diaphora

TABLE 4. Summary of comparison for x86_64 architecture compiled unstripped binaries.

Software	Tool	Time (sec)	Matches TP	Matches FP	Unmatched
xnu-4570-31.1 ⊖ xnu-4570-41.1	Diaphora	861.1	17042 (**96%**)	694	15
	Reveal	73604	15407 (84.80%)	79	2681
	Proposed	**353.25**	17215 (94.60%)	745	236
xnu-4570-41 ⊖ xnu-4570-51	Diaphora	798.14	13761 (77.87%)	274	3635
	Reveal	47018	13344 (73.43%)	1833	2994
	Proposed	**750**	14910 (**82.04%**)	2860	402
xnu-4570-51.1 ⊖ xnu-4570-61.1	Diaphora	755.67	17017 (**95.87%**)	703	29
	Reveal	21980	15479 (84.90%)	102	2649
	Proposed	**667**	17211 (95.02%)	520	381
xnu-4570-61.1 ⊖ xnu-4570-71.1	Diaphora	872.01	16456 (92.71%)	282	1011
	Reveal	24748	15420 (84.52%)	141	2674
	Proposed	**357.31**	17199 (**95.07%**)	517	375

D. x86_64 ARCH BINARIES–UNSTRIPPED

In this section, we have evaluated the function matching tools' performance over XNU kernel binaries, which are manually compiled for x86 architecture. For unstripped binaries, symbolic information is available thus features based on such information can be more effective. Diaphora features set to utilize the symbolic information but in contrast, Reveal and the proposed tool useless or do not use such information. The comparison results for unstripped binaries are reported in Table 4. For the first and third experiments, Diaphora TPs percentage is higher than the compared tools. This improvement in TPs is first linked with the availability of debug symbols and secondly, there is less number (<60) of partially changed functions in compared binaries. In the second and fourth experiments, the number of parts changed functions are more than 500, and Diaphora heuristics (based on a single feature) mismatches such partially changed functions, which result in a drop in performance.

Reveal performance is comparatively less in all scenarios but consistent. Its FMA fails to match at least 15% of functions as shown in the unmatched column of Table 4. In the first and third experiments, the proposed tool efficiency is comparable to Diaphora but for the second and fourth experiments, proposed tool outperforms Diaphora. One-shot encoding mechanism and distance-based criteria (adopted in the proposed tool) can accurately match partially changed function as compared to Diaphora. The proposed tool is consistent in its performance against all experiments but Diaphora is not consistent. The proposed tool is also computationally efficient as compared to baselines. Although the proposed

TABLE 5. Summary of comparison for x86_64 architecture compiled stripped binaries.

Software	Tool	Time (sec)	Matches TP	Matches FP	Unmatched
xnu-4570-31.1 ⊖ xnu-4570-41.1	Diaphora	818.4	16735 (95.23%)	827	11
	Reveal	75535	15472 (85.17%)	13	2681
	Proposed	**368**	16842 (**96.88%**)	342	199
xnu-4570-41 ⊖ xnu-4570-51	Diaphora	906.4	13696 (77.51%)	271	3702
	Reveal	46921	13639 (75.06%)	1537	2994
	Proposed	**353.25**	14868 (**86.51%**)	2015	301
xnu-4570-51.1 ⊖ xnu-4570-61.1	Diaphora	785.7	16852 (95.27%)	806	29
	Reveal	21663	15550 (85.30%)	30	2649
	Proposed	**412.5**	16845 (**97.41%**)	179	268
xnu-4570-61.1 ⊖ xnu-4570-71.1	Diaphora	862.4	16447 (92.66%)	1178	124
	Reveal	24341	15521 (85.11%)	49	2665
	Proposed	**410.2**	16826 (**97.55%**)	155	267

tool is efficient in complex cases, we also endorse Diaphora for matching unstripped x86 compiled binaries.

In brief, when the compared binaries have less number of partially changed functions and binaries are unstripped; Diaphora is efficient in comparing exact functions but the proposed tool is efficient in comparing exact and partially changed functions.

E. x86_64 ARCH BINARIES–STRIPPED

In this section, we have evaluated the function matching tools' performance over XNU kernel binaries, which are officially distributed x86 binaries. The officially released binaries are stripped and most of (40-60%) the symbolic information is not available. The comparison results for the stripped binaries are reported in Table 5. Although Reveal performance is comparatively less among other tools, its performance is similar to unstripped binaries. In other words, reveal features are not dependent on symbolic information thus its performance is not affected.

Some heuristics in Diaphora are dependent on symbolic information thus in comparison to stripped binaries, the TPs accuracy slightly drops for the stripped binaries. Comparing Table 4 and Table 5 results for Diaphora, the number of TPs reduced and number of FPs increased. In contrast, the proposed tool is consistent with Reveal in its detection accuracy. There are two observations from Table 5 results. (i) There is a difference in the total number of functions for Diaphora, Reveal vs proposed tool. This difference is due to the reason discussed in section V-C. For the fair evaluation, we have used the relative percentage for TP. In comparison to Diaphora, the proposed tool has better detection accuracy for all four experiments for stripped binaries. (ii) Diaphora and the proposed tool both have a higher number of FPs as compared to Reveal. Reveal only focuses on finding the exact matches. However, there are partially changed functions as well, which must also be mapped as matched. While matching partially changed functions, the probability of false positives rises as the FMA is a feature-based approximation that is more accurate for exact matches with zero distance but increasing the threshold to a higher number introduces inaccuracies. Although considering partially changed functions increases FPs for Diaphora and proposed tools but as an advantage TPs percentage is increased.

It can be concluded that the proposed features are efficient for x86 stripped binaries and Diaphora is the second-best option. For x86 architecture, detection accuracy for Diaphora and proposed tools are fairly efficient.

TABLE 6. Summary of comparison for ARM architecture.

Software	Tool	Time (sec)	Matches TP	Matches FP	Unmatched
xnu-4570-31.1 ⊖ xnu-4570-41.1	Diaphora	**2060**	40154 (97.05%)	38	1179
	Reveal	12176	36833 (87.72%)	0	5155
	Proposed	2265.125	35002 (**97.84%**)	414	51
xnu-4570-41 ⊖ xnu-4570-51	Diaphora	2634.3	21941 (52.74%)	563	19092
	Reveal	152000	35582 (**84.70%**)	333	6091
	Proposed	**2430**	25509 (71.85%)	1581	8410
xnu-4570-51.1 ⊖ xnu-4570-61.1	Diaphora	**2022**	29074 (69.96%)	425	12055
	Reveal	18461	36869 (87.51%)	9	5252
	Proposed	2290	34740 (**97.53%**)	703	170
xnu-4570-61.1 ⊖ xnu-4570-71.1	Diaphora	4163	34264 (82.38%)	293	7033
	Reveal	14775	36966 (87.66%)	0	5199
	Proposed	**2295.4**	34652 (**97.06%**)	811	233

TABLE 7. Results for searching the heartbeat vulnerability.

From → To	discovRE TLS	discovRE DTLS	VSkLCG TLS	VSkLCG DTLS	Proposed TLS	Proposed DTLS
X86 → ARM	1	1	1	1	1	1
ARM → X86	1	1	1	1	1	1

F. ARM ARCH BINARIES

In this section, we have evaluated the function matching tools' performance over XNU kernel binaries, which are compiled for the ARM architecture. Different architecture binaries differ only in their textual representation i.e. assembly code but the structural features are almost the same. Reveal textual representation features do not use assembly code as a feature thus its performance is consistent for TPs.

TABLE 8. Cross-optimization-level binary comparison results.

	Recall				Precision			
	BinDiff	Asm2Vec+k-Hop	DeepBinDiff	Proposed	BinDiff	Asm2Vec+k-Hop	DeepBinDiff	Proposed
v5.93 O0-O3	0.176	0.155	0.311	**0.530**	0.291	0.211	0.315	**0.395**
O1-O3	0.571	0.545	0.666	**0.910**	0.638	0.544	**0.681**	0.635
O2-O3	0.837	0.911	**0.975**	0.950	0.944	0.859	**0.955**	0.809
v6.4 O0-O3	0.166	0.201	0.391	**0.493**	0.262	0.235	0.301	**0.398**
O1-O3	0.576	0.579	0.703	**0.904**	0.646	0.563	**0.709**	0.634
O2-O3	0.838	0.893	0.949	**0.956**	0.947	0.851	**0.953**	0.811
v7.6 O0-O3	0.156	0.255	0.301	**0.440**	0.331	0.291	**0.40**	0.328
O1-O3	0.484	0.618	0.672	**0.908**	0.674	0.599	**0.761**	0.583
O2-O3	0.840	0.903	0.946	**0.954**	**0.942**	0.861	0.941	0.751
v8.1 O0-O3	0.166	0.169	0.305	**0.447**	0.334	0.291	**0.351**	0.342
O1-O3	0.480	0.625	0.679	**0.913**	0.677	0.612	**0.721**	0.590
O2-O3	0.835	0.871	0.915	**0.949**	**0.942**	0.828	0.912	0.753
v8.30 O0-O3	0.135	0.144	0.292	**0.403**	0.285	0.275	0.315	**0.327**
O1-O3	0.508	0.521	0.602	**0.841**	0.620	0.493	**0.665**	0.593
O2-O3	0.842	0.875	**0.956**	**0.930**	0.954	0.843	0.908	0.750
Average	0.507	0.549	0.65	**0.769**	0.632	0.557	**0.659**	0.580

As a drawback, its feature set is not diverse and consequently, FMA cannot match all functions and the number of unmatched functions is increased. In contrast, Diaphora uses assembly code as a feature but it is fine-tuned only for x86 architecture. Consequently, its function matching accuracy is severely affected for the ARM architecture. As shown in Table 6, Diaphora achieves higher matching accuracy for the first experiment but its accuracy drops for the other three experiments. In brief, Diaphora features are optimized for x86 architecture only, and for ARM architecture its accuracy is abrupt and inconsistent.

Proposed tool textual features are based on intermediate representation (ESIL), which is the same for all architectures, Hence, the proposed features are resilient against multi-architectures; as compared to Diaphora. Textual representation-based features cannot be ignored like the Reveal approach as they are as important as structural representation-based features. Diversity and uniqueness of features are the keys to accurately match functions. Proposed features are not architecture-dependent and the proposed FMA achieves higher function matching accuracy for the ARM architecture; as shown in Table 6.

The proposed tool is consistent in its accuracy against evaluated architectures, whereas the accuracy of Reveal and Diaphora drop significantly for the ARM architecture. It can be concluded that the proposed features are efficient and expandable to multi-architectures.

G. CROSS ARCHITECTURES COMPARISON

In this section, we have evaluated the performance of the proposed tool for cross-architecture comparison. DiscovRE [8] and VSkLCG [16] have proposed function matching techniques that compare the cross-architecture binary executables. Our work is not fine-tuned for cross-architecture comparisons but rather is a general solution for multi-architectures binary comparison. However, we have evaluated the performance of our tool for cross-architecture binaries, and results are reported in Table 7. This section experiments were conducted on the same configurations like discovRE and OpenSSL binaries were used for comparison, which have two vulnerable functions i.e. *tls1_process_heartbeat* and *dtls1_process_heartbeat*. The evaluation objective is to search these two vulnerable functions in cross architectures and output as *1: 'detected', 0:'not detected'*. Evaluations in DiscovRE have compared three architectures *ARM, X86, MIPS* but unfortunately we could not compare *MIPS*; due to limitations of Rardare2 (underlying binary analysis tool) as it does not support MIPS at the time of writing.

As shown in Table 7, all tools can detect the vulnerable functions. In the experiments, we have considered $k = 5$ in the KNN algorithm to report the vulnerable functions; whereas DiscovRE uses the $k = 128$ and selects the top 5% to report the vulnerable functions. In brief, it can be concluded that the proposed tool can accurately match the cross-architecture binaries when $k = 5$ and matched functions are in the top 5 results. However, discovRE performance is better than the proposed tool in the top 1% and we equally endorse discovRE for cross-architecture comparison.

H. RESILIENCE AGAINST COMPILER OPTIMIZATIONS

In this section, we have evaluated the resilience against the optimization-level settings. We have utilized the DeepBinDiff [3] open-sourced binaries,[8] which were compiled for O0-O3, O1 vs O3, O2 vs O3. The evaluation parameters are recall and precision and results are reported in Table 8.

In Table 8, the results for *BinDiff* [5], *Asm2Vec+k-Hop*, and *DeepBinDiff* are reported from the DeepBinDiff [3] paper. The results for proposed are computed with Algorithm 4, using the same experimental configurations like DeepBinDiff. For evaluation of results, we compare each binary in *coreutils*, compiled with optimization level O3; with

[8]https://github.com/yueduan/DeepBinDiff.git

the same version Coreutil binaries compiled with optimization levels O0, O1, and O2, respectively. Each version of Coreutils consists of around 80-110 binaries, we first compare them individually and compute TP, FP, FN and later take an average of TP, FP, and FN to compute recall and precision. As shown in Table 8, the proposed tool outperforms in recall compare to other tools but DeepBinDiff mostly outperforms in precision. The reason for this behavior in the proposed tool is tightly linked with the embedding vector-based function modeling. The embedding vector modeling in section III-G2, suppresses the structural changes caused by compiler optimizations, which improves the recall but it replaces some assembly tokens with fixed constants that affect the precision in the function matching algorithm. In contrast, DeepBinDiff uses the *softmax* based classifier to learn the embedding representation and merges it with CFG, thus preserving the structural information. Consequently, it has high precision but low recall in its function matching algorithm.

In brief, it can be concluded that the proposed function modeling-based function vector feature can handle the compiler optimization to some degree and can be effectively used as a function feature. However, we endorse DeepBinDiff for the high precision function matching tasks.

VI. CONCLUSION AND LIMITATIONS

Diaphora has a diverse set of features and their function matching algorithm is excellent for unstripped binaries, which are compiled for x86 architecture. Diaphora's best heuristics rely on textual representation-based features but these features are not optimal when the binaries are stripped. Evaluation results show that Diaphora slightly loses its efficiency for stripped x86 binaries. For ARM architecture, Diaphora is unstable and inaccurate as its function matching algorithm cannot match 20-30% functions.

Reveal has less feature set as compared to Diaphora but its function matching accuracy is consistent for stripped and multi-arch binaries. There are two drawbacks in Reveal, first, it is not computationally efficient for large binaries and second, its matching accuracy is less than Diaphora and the proposed tool. The drop inefficiency is due to its misinterpretation of the function matching output. Reveal only considers the exact matches as the output of FMA but the reality is that partially changed functions must also be 1:1 mapped. Although Reveal proposed features are efficient but considering only the exact matches, it loses its efficiency among compared tools.

The proposed tool uses efficient features with a one-shot encoding mechanism that boosts its computational efficiency and also its detection accuracy is consistent for multi-architectures. Evaluation results show that using intermediate representation; in modeling textual representation-based features, is the main reason for consistency in multi-architectures and cross-optimization levels. The distance-based selection criteria are flexible to partially changed function but it also introduces inaccuracies for some special scenarios: given below. Although our technique is effective against multi-architectures, it has the following two limitations.

1) In our empirical analysis, we have found that the most TP cases detected by the proposed tool are the ones that have less than 5 assembly instructions; defined in a single basic block. For a single basic block, proposed features are not effective thus proposed tool mismatches such functions with other similar functions that also have a single basic block. To explore the solution, we have investigated the Diaphora source code. But to our surprise, if the number of instructions is less than 10 then its FMA ignores such functions. The assembly code of functions with less than 5 instructions looks very similar to each other and hard to accurately match them. In the future, we aim to further investigate this challenge and explore a feasible solution.
2) Proposed textual representation features are modeled with *word2vec* based embedding vectors. Evaluation results show that this technique is effective but if there are unseen strings then the word2vec model needs to be retrained. This problem is general to all embedding-based techniques. To cater to this problem, we aim to split the multi-token strings to a *list* of tokens and formulate an equivalent vector by the summation of their embedding vectors. A similar approach is also adopted by Google trained word2vec model and its effectiveness against unseen strings dataset. We will implement this feature in our tool, before the final release.

APPENDIX A
ANALYSIS TOOLS VS IR CHOICE

Given two binary executables, function matching requires the pre-analysis of binaries for the availability of each function representation i.e. CFG and assembly code. Binary analysis tools lay the foundation for pre-analysis of binaries by providing parser for binary formats (ELF, Mach-O, PE), relocations analyzer, function identification analyzer, disassembler, static analysis (CFG) utility, etc. The details of each step are out of the scope of this research but a step-by-step process of a binary analysis tool is shown in Figure 11.

There are many binary analysis tools that differ in the scope of their functionality. An objective of this research is to provide a multi-arch solution for function matching, so a binary diffing tool must be efficient in providing base functionality for well-known architectures and file formats. We have analyzed Angr, BAP, IDA, and Radare2, which are frequently used in literature research. In our empirical analysis, we have explored that all of these tools are matured for PE and ELF binary formats but their accuracy drops for Mach-O format. There are several errors in file parsing or function analysis modules. As shown in Table 9, Radare2 is best among other binary analysis tools and it analyzes all format binaries with the same accuracy. IDA pro is the second-best choice (satisfying all parameters) and also an industry-standard tool but it does not provide IR lifting thus cannot serve the purpose

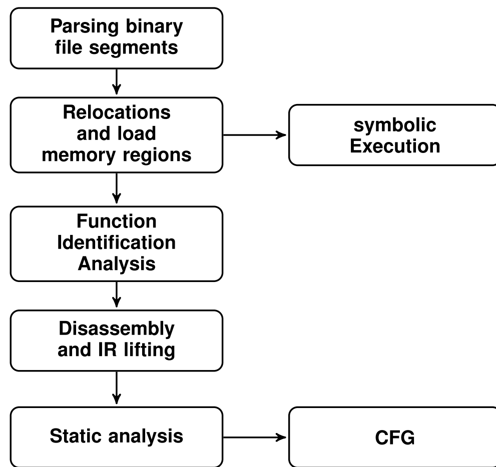


FIGURE 11. A general binary analysis procedure.

of this research. Radare2 being a single choice is selected for this research.

TABLE 9. Comparison of binary analysis tools.

Parameters	Angr	BAP	IDA	Radare2
IR language	Vex	BIL.	REIL	ESIL
File Parsing	✗	✓	✓	✓
Memory Loading	✓	✓	✓	✓
Function Identification	✗	✓	✓	✓
Disassembly	✓	✓	✓	✓
IR lifting	✗	✗	✗	✓
CFG construction	✗	✓	✓	✓
Built-in Static analysis	✓	✓	✓	✓

APPENDIX B
ESIL: Radare2 INTERMEDIATE REPRESENTATION LANGUAGE

Evaluable Strings Intermediate language (ESIL) is an IR language created and used by Radare2. To represent IR statements, it utilizes a post-fix notation that is stack friendly. A value is simply pushed on the stack, an operator then pops values from the stack, performs op-code operation and pushes the result back on.

An ESIL transformation machine uses the following rules to translate assembly instructions to ESIL expressions.

- A target assembly opcode is translated into a comma separated list of ESIL expressions. $xor \quad eax, eax \rightarrow 0, eax, =, 1, zf$
- Memory access is defined by brackets operation. E.g. $mov \quad eax, [0x80480] \rightarrow 0\,x80480, [], eax, =$
- Default operand size is determined by the size of operation destination. $movb \quad \$0, 0x80480 \rightarrow 0, 0x80480, = []$
- NOP instructions is represented as an empty string.
- syscalls are marked by $ sign. E.g. $0\,x80, \$$

A. ESIL INSTRUCTION SET

Radare2 ESIL defines its own instruction set using Table 10 given opcodes. These opcodes are applicable to all architectures and serve the purpose for an intermediate representation language.

TABLE 10. ESIL opcodes.

Opcode type	Index	Opcode type	Index
NULL	0	SAR	23
JMP	1	OR	24
UJMP	2	AND	25
CJMP	2	XOR	26
CALL	3	NOR	27
UCALL	4	NOT	28
RET	5	STORE	29
CRET	5	LOAD	30
NOP	6	LEA	31
MOV	7	INVALID	32
CMOV	7	ROR	33
TRAP	8	ROL	34
SWI	9	ILL	35
CSWI	9	UNK	36
UPUSH	10	XCHG	37
PUSH	11	MOD	38
POP	12	LEAVE	39
CMP	13	SWITCH	40
ACMP	14	CASE	41
ADD	15	LENGTH	42
SUB	16	CAST	43
IO	17	NEW	44
MUL	18	ABS	45
DIV	19	CPL	46
SHR	20	CRYPTO	47
SHL	21	SYNC	48
SAL	22		

REFERENCES

[1] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digit. Invest.*, vol. 12, pp. S61–S71, Mar. 2015.
[2] S. Alrabaee, L. Wang, and M. Debbabi, "BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs)," *Digit. Invest.*, vol. 18, pp. S11–S22, Aug. 2016.
[3] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020.
[4] H. Flake, "Structural comparison of executable objects," in *Proc. DIMVA*, Dortmund, Germany, Jul. 2004, pp. 161–173.
[5] *Zynamics BinDiff Kernel Description*. Accessed: Feb. 10, 2020. [Online]. Available: https://www.zynamics.com/bindiff.html
[6] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 266–280, 2016.
[7] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (English version)," in *Proc. SSTIC*, 2005, vol. 5, no. 1, p. 3.
[8] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Proc. NDSS*, 2016, pp. 58–79, doi: 10.14722/ndss.2016.23185.
[9] C. Karamitas and A. Kehagias, "Efficient features for function matching between binary executables," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2018, pp. 335–345, doi: 10.1109/SANER.2018.8330221.
[10] *Diaphora Ida Plugin*. Accessed: Feb. 10, 2020. [Online]. Available: https://github.com/joxeankoret/diaphora
[11] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "FOSSIL: A resilient and efficient system for identifying FOSS functions in malware binaries," *ACM Trans. Privacy Secur.*, vol. 21, no. 2, pp. 1–34, Feb. 2018.

[12] J. Oh, "Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries," in *Proc. Blackhat Tech. Secur. Conf.*, Jul. 2009.
[13] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *Proc. Int. Conf. Inf. Commun. Secur.* Berlin, Germany: Springer, 2008, pp. 238–255.
[14] Z. Wang, K. Pierce, and S. McFarling, "BMAT—A binary matching tool for stale profile propagation," *J. Instruct.-Level Parallelism*, vol. 2, pp. 1–20, May 2000.
[15] Z. Wang, K. Pierce, and S. McFarling, "BMAT—A binary matching tool," in *Proc. 2nd ACM Workshop Feedback-Directed Optim.*, 1999, pp. 1–20.
[16] M. Han, D. Zhao, H. Lin, D. Zhou, J. Xiang, Z. Liu, and Y. Xing, "VSkLCG a method for cross-platform vulnerability search in firmware," in *Proc. 6th Int. Conf. Dependable Syst. Their Appl. (DSA)*, Jan. 2020, pp. 395–400.
[17] M. Bourquin, A. King, and E. Robbins, "BinSlayer: Accurate comparison of binary executables," in *Proc. 2nd ACM SIGPLAN Program Protection Reverse Eng. Workshop*, 2013, pp. 1–10, doi: 10.1145/2430553.2430557.
[18] C. Karamitas and A. Kehagias, "Function matching between binary executables: Efficient algorithms and features," *J. Comput. Virol. Hacking Techn.*, vol. 15, no. 4, pp. 307–323, Dec. 2019.
[19] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proc. ACM Program. Lang.*, vol. 3, 2019, pp. 1–29, doi: 10.1145/3291636.
[20] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 480–491.
[21] T. Dullien, E. Carrera, S.-M. Eppler, and S. Porst, "Automated attacker correlation for malicious code," Bochum Univ., Bochum, Germany, Tech. Rep. RTO-MP-IST-091, 2010. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.661.9484&rep=rep1&type=pdf
[22] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proc. 18th Int. Symp. Softw. Test. Anal.*, 2009, pp. 117–128.
[23] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proc. 8th Work. Conf. Mining Softw. Repositories*, 2011, pp. 63–72.
[24] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-architecture cross-OS binary search," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2016, pp. 678–689.
[25] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 709–724.
[26] M. R. Farhadi, B. C. M. Fung, P. Charland, and M. Debbabi, "BinClone: Detecting code clones in malware," in *Proc. 8th Int. Conf. Softw. Secur. Rel. (SERE)*, Jun. 2014, pp. 78–87.
[27] M. R. Farhadi, B. C. M. Fung, Y. B. Fung, P. Charland, S. Preda, and M. Debbabi, "Scalable code clone search for malware analysis," *Digit. Invest.*, vol. 15, pp. 46–60, Dec. 2015.
[28] S. Cesare, Y. Xiang, and W. Zhou, "Control flow-based malware variant detection," *IEEE Trans. Depend. Sec. Comput.*, vol. 11, no. 4, pp. 307–317, Jul./Aug. 2013.
[29] X. Hu, T.-C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 611–620.
[30] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αDiff: Cross-version binary code similarity detection with DNN," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, Sep. 2018, pp. 667–678, doi: 10.1145/3238147.3238199.
[31] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," 2018, *arXiv:1808.04706*. [Online]. Available: http://arxiv.org/abs/1808.04706
[32] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 363–376, doi: 10.1145/3133956.3134018.
[33] K. Redmond, L. Luo, and Q. Zeng, "A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis," 2018, *arXiv:1812.09652*. [Online]. Available: http://arxiv.org/abs/1812.09652
[34] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proc. 26th USENIX Secur. Symp. (USENIX Security)*, 2017, pp. 99–116.
[35] S. Ullah and H. Oh, "BinDiff$_{NN}$: Learning distributed representation of assembly for robust binary diffing against semantic differences," *IEEE Trans. Softw. Eng.*, early access, Jul. 1, 2021, doi: 10.1109/TSE.2021.3093926.
[36] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2702–2711.
[37] R. Tarjan, "Testing flow graph reducibility," in *Proc. 5th Annu. ACM Symp. Theory Comput.*, 1973, pp. 96–107.
[38] Y. Li, J. Jang, and X. Ou, "Topology-aware hashing for effective control flow graph similarity analysis," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Cham, Switzerland: Springer, 2019, pp. 278–298.
[39] M. Slaney and M. Casey, "Locality-sensitive hashing for finding nearest neighbors [lecture notes]," *IEEE Signal Process. Mag.*, vol. 25, no. 2, pp. 128–131, Mar. 2008.
[40] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proc. 20th Annu. Symp. Comput. Geometry*, 2004, pp. 253–262.
[41] Radare2 Team, GitHub, 2017. [Online]. Available: https://github.com/radareorg/radare2-book and https://book.rada.re/
[42] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *Proc. KSII 9th Int. Conf. Internet Symp. (ICONI)*, 2017, pp. 1–5.
[43] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489, doi: 10.1109/SP.2019.00003.
[44] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*. [Online]. Available: http://arxiv.org/abs/1301.3781
[45] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 1188–1196.

SAMI ULLAH received the B.S. degree in electronics engineering from International Islamic University Islamabad, Pakistan, in 2014. He is currently pursuing the M.S./Ph.D. degree with the Department of Computer Science and Engineering, Hanyang University, South Korea. From 2015 to 2017, he worked as a Teaching and Research Assistant with the Next Generation Intelligent Networks Research Center, Institute of Space and Technology. He is also working as a Research Assistant with InfoSec Lab. His research interests include robust network security, vulnerability analysis at application layer and network layer, and binary static analysis using machine learning and deep learning techniques.

WENHUI JIN was born in Hegang, Heilongjiang, China. He received the B.S. degree in software and engineering from Heilongjiang University, China, in 2013. He is currently pursuing the Ph.D. degree in computer science and engineering with Hanyang University, South Korea. His research interests include binary obfuscation, binary analysis with machine learning, malware analysis, and detection, and security issues in mobile.

HEEKUCK OH (Member, IEEE) received the B.Sc. degree in electronics engineering from Hanyang University, in 1983, and the M.S. and Ph.D. degrees in computer science from Iowa State University, in 1989 and 1992, respectively. He joined the Faculty of the Department of Computer Science and Engineering, Hanyang University, Erica campus, in 1994, where he is currently a Professor. He is also the President Emeritus with Korea Institute of Information Security and Cryptography. His current research interests include network and system security. He is also a member of the Advisory Committee for Digital Investigation in the Supreme Prosecutors' Office of the Republic of Korea. He is also a member of the Advisory Committee on Government Policy under the Ministry of Government Administration and Home Affairs.