How Machine Learning Is Solving the Binary Function Similarity Problem

Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, Davide Balzarotti

How Machine Learning Is Solving the Binary Function Similarity Problem

Andrea Marcelli
Cisco Systems, Inc.

Mariano Graziano
Cisco Systems, Inc.

Xabier Ugarte-Pedrero
Cisco Systems, Inc.

Yanick Fratantonio
Cisco Systems, Inc.

Mohamad Mansouri
EURECOM

Davide Balzarotti
EURECOM

Abstract

The ability to accurately compute the similarity between two pieces of binary code plays an important role in a wide range of different problems. Several research communities such as security, programming language analysis, and machine learning, have been working on this topic for more than five years, with hundreds of papers published on the subject. One would expect that, by now, it would be possible to answer a number of research questions that go beyond very specific techniques presented in papers, but that generalize to the entire research field. Unfortunately, this topic is affected by a number of challenges, ranging from reproducibility issues to opaqueness of research results, which hinders meaningful and effective progress.

In this paper, we set out to perform the first measurement study on the state of the art of this research area. We begin by systematizing the existing body of research. We then identify a number of relevant approaches, which are representative of a wide range of solutions recently proposed by three different research communities. We re-implemented these approaches and created a new dataset (with binaries compiled with different compilers, optimizations settings, and for three different architectures), which enabled us to perform a fair and meaningful comparison. This effort allowed us to answer a number of research questions that go beyond what could be inferred by reading the individual research papers. By releasing our entire modular framework and our datasets (with associated documentation), we also hope to inspire future work in this interesting research area.

1 Introduction

Binary function similarity is the problem of taking as input the binary representation of a pair of functions, and producing as output a numeric value that captures the "similarity" between them. This problem is very challenging to solve in the general case. In fact, software is often compiled with different toolchains, different compiler optimizations and flags, and, in some scenarios like IoT devices, software is compiled to different architectures, making trivial binary similarity approaches ineffective.

Binary function similarity plays a crucial role in different systems security research fields, as a number of research problems require measuring function similarity as a core building block. For example, reverse engineers often deal with stripped binaries that have been statically linked (thus without symbols), and binary code similarity approaches can be used to match an unknown function to (labeled) functions in a previously generated database, saving numerous hours of reverse engineering effort. Binary similarity is also crucial to effectively detect and patch vulnerabilities in third-party libraries. In this case, given a vulnerable function, similarity techniques help finding occurrences of that same function in one or more binaries, allowing for a much quicker identification and patching of problematic bugs. As additional examples, this problem is also relevant for binary diffing and patch analysis, in which two binaries with multiple functions must be compared against each other, and in software lineage analysis or malware clustering, in which the analysts is interested in pinpointing common functions across different malware samples and to group them together according to their similarity.

The importance and relevance of this problem is reflected by the available literature: researchers in many different disciplines, including systems security, programming languages, and machine learning, have published an astonishing number of papers (often in their respective top venues) to propose new approaches for binary similarity. This competition has resulted in a very rapid evolution of the existing techniques, and in the progressive development and refinement of multiple solutions.

One would expect that this significant body of work would be sufficient to answer a number of important research questions. For example: How do different approaches compare when evaluated with the same dataset and by using the same metrics? Which are the main contributions of the novel machine-learning solutions compared to simpler fuzzy hashing approaches? Which is the role of different sets of features? Do different approaches work better at different tasks? Is the cross-architecture comparison more difficult

to solve than working on a single architecture? Is there any specific line of research that looks more promising as a future direction for designing new techniques? Unfortunately, we found that the current body of published research is unable to answer these questions, due to a number of major challenges.

Challenges. The first challenge is the current inability to neither reproduce nor replicate previous results. While this is sadly a common problem in the security field, the area of binary similarity is a particularly good example of this issue. Only 12 out of the 61 solutions reported in the survey by Haq et al. [27] released their tool to other researchers. And even when the artifacts are available, they are often incorrect (i.e., they do not implement the exact same solution described in the paper), incomplete (i.e., important components, for instance for features extraction, are missing), or the code might not even run on datasets different from the one used by its authors. Since re-implementing previous techniques is very complex and extremely time consuming, each solution is typically compared with only a couple of previous techniques that may sometimes not even be designed to solve the same problem (e.g., code search vs. binary diffing), and in some extreme cases the comparison is done only against a previous paper from the same authors. The lack of reproducibility is even more relevant for machine-learning approaches, where implementation choices, hyperparameters, and training and testing methodologies strongly influence the results. It is also often unclear whether the released models should be used as-is or whether retraining is necessary to reproduce similar results on different datasets.

The second challenge is that the evaluation results are often opaque. Different solutions are typically customized for slightly different objectives (e.g., searching for a vulnerability vs. finding similar malware samples), in different settings (e.g., cross-compiler vs. cross-architecture), by using a different concept of similarity (same code vs. same semantic), and operating at different granularities (e.g., code snippets vs. entire functions). The experiments are also performed on datasets of different size and nature (e.g., firmwares vs. command-line utilities), and the results are reported by using different metrics (e.g., ROC curves vs. top-n vs. MRR10). Therefore, even the most basic figures reported in each paper are not directly comparable. Thus, when results outperform previous works, it is unclear whether it happens only in the selected scenario or also in other use cases. To make things worse, papers often omit details on how functions are filtered out and how positive and negative pairs are selected for training, making it difficult to reproduce the pipeline faithfully even with the same binaries.

Note also that these works are often built on top of non-trivial pipelines, e.g., toolchains to determine function boundaries, disassemble the code, and extract the control-flow graph. The few available approaches use different toolchains and they are built on different pipelines. It is thus very challenging to determine how much the reliability of the initial "non-relevant" stages of the pipeline actually affects the reliability of the overall approach. In other words, it is often unclear whether the superior results of a given approach are related to the contributions presented as novel or are instead related to other factors.

The combined effect of the first two challenges resulted in a field that is extremely fragmented, where dozens of techniques exist but without a clear understanding of what works (or does not) in which settings. This brings us to the last challenge: the difficulty of understanding which direction binary similarity research is heading, and why. Each new solution adopts a more complex technique, or a new combination of multiple techniques, and it is difficult to tell whether this is driven by actual limitations of the simpler approaches or by the need to convince the reviewers about the novelty of each work. This fragmentation has often led to *parallel and disjoint lines of research*, where everyone is claiming to have the best solution. This fragmentation has also led to papers with sub-optimal evaluations and approaches. For example, papers that are strong on the program analysis aspect may lack the application of state-of-the-art machine-learning techniques. Solutions based on machine learning are the current trend, but they often blindly apply techniques from other fields, making it harder to judge the overall progress and innovation in the area.

Contributions. In this paper, we perform the first systematic measurement in this area of research. We first explore existing research and group each solution based on the adopted approach, with a particular focus on recent successful techniques based on machine learning. We then select, compare, and *implement* the ten most representative approaches and their possible variants. These approaches are representative of a wide range of trends and span across three different research communities: the computer security, the programming language analysis, and the machine-learning communities. To make our comparisons meaningful, our implementations are built on top of a common framework (e.g., we extract the raw features using the same underlying implementation, while previous works rely on different toolchains). If the original implementation is available, we include the core model implementation in a common codebase for training and testing and we extend the support for missing architectures and bitnesses. Finally, we leverage parallel programming and efficient data encoding techniques to avoid bottlenecks that could negatively affect the model performances.

By re-implementing various *approaches*—and not necessarily the "papers"— we isolate existing "primitives" and evaluate them when used independently or combined with each other, to gain insights and pinpoint important factors that are hidden in the complexity of previous works, and to answer various open research questions. To make this evaluation effort more comparable, we also propose a new dataset that we use as a common benchmark with varying aspects such as compiler family, optimizations, and architectures.

Note that our research focuses on evaluating the main techniques proposed to date without trying to reproduce the exact results reported in the corresponding papers. While some of our implementations are derived from the code released by the authors when available, others have been developed from

scratch with the goal of having a single codebase and pipeline that can isolate the technique from the rest of factors that can influence the result.

Our evaluation highlights several interesting insights. For example, we found that while simple approaches (e.g., fuzzy hashing) work well for simple settings, they fail when dealing with more complex scenarios (such as cross-architecture datasets, or datasets for which multiple variables change at the same time). Among the machine-learning models, those based on Graph Neural Network achieved the best results in almost all the tasks, and are among the fastest when comparing the inference time. Another interesting finding is that many recently published papers all have very similar accuracy when tested on the same dataset, despite several claims of improvement over the state of the art.

While we do not claim that our code or dataset is better or more representative than previous works, *we release our modular framework, the re-implementation of all the selected approaches, the full dataset, and detailed instructions on how to recreate it and tweak it.*[1] By allowing the community to experiment with the individual components and to directly compare one against each other, we hope to encourage and ease the effort of future researchers that are interested in approaching this active research area.

2 The Binary Function Similarity Problem

In its simplest form, binary function similarity aims at computing a numeric value that captures the "similarity" between a pair of functions in their binary representation, raw bytes (i.e., machine code) constituting the body of the function, as produced by a compiler. Note that, while in this paper we focus on approaches that use functions as units of code, researchers have also studied techniques that focus on lower-level abstractions (e.g., basic blocks) or higher-level ones (e.g., whole programs). The term *similarity* has instead various interpretations, depending on the context. For this paper, we consider two functions as "similar" if they have been compiled from the same source code, independently from the compiler, its version, its compilation flags, or even the architecture the function has been compiled to (e.g., x86, ARM). Thus, according to our definition, two "similar" functions may have vastly different binary representations — and this is what makes this research problem interesting and challenging.

Binary function similarity has been studied in more than a hundred papers. To complicate the landscape, most of the existing approaches cannot be mapped to a single category of techniques, as they are often built on top of different components. Therefore, in this section we focus on the different building blocks that these approaches are composed of, by looking first at the techniques to compute similarity, and then at the types of input data that these approaches can make use of.

[1] All our artifacts and additional technical information are available at https://github.com/Cisco-Talos/binary_function_similarity, referred throughout the paper as [47].

2.1 Measuring Function Similarity

Direct vs. indirect comparison. We can group the techniques to measure function similarity in two main categories. The first class of solutions implement a *direct comparison* of pairs of functions, either by considering raw input data or by implementing some sort of feature extraction. These solutions often need to *learn* that two seemingly-unrelated values can represent similar functions, or vice-versa that close values do not necessarily represent something similar. This is the case when the features extracted from binary functions cannot be directly compared by using basic similarity metrics as they may not be represented in a linear space, or may not have an equivalent weight on the similarity score. Therefore, researchers have proposed to use machine-learning models in order to determine if two functions are similar given a set of extracted features as input. There are several approaches that implement this type of similarity by leveraging Bayesian networks [2], convolutional neural networks [44], Graph Matching Networks (GMN) [40], regular feed-forward neural networks [67], or combinations of them [37]. In these cases, the model is used to output a similarity score between a pair of functions.

To find similar functions, these approaches need to search over the entire dataset and compare the features of the queried function against every entry in the dataset, which is not a scalable solution. For this reason, many approaches implement indexing strategies to pre-filter potentially similar candidates with techniques such as tree-based data structures [55, 68], locality sensitive hashing [15, 22, 32, 56, 61] (approximate nearest neighbor search), bloom filters [35], custom pre-filters based on more simple numeric data [6, 20], clustering techniques [81], or even distributed search approaches such as map-reduce [15].

The second class of solutions implement *indirect comparison* techniques. These approaches map the input features to a "condensed" lower-dimensional representation that can be easily compared to one another using a distance measure, like the euclidean or the cosine distance. These solutions allow efficient one-to-many comparisons. For instance, if a new function needs to be compared against an entire dataset, one can first map each function in the repository to its respective low-dimension representation (this is a one-off operation), then perform the same operation on the new function, and finally compare these representations by using efficient techniques such as approximate nearest-neighbors.

Fuzzy hashes and embeddings. A popular example of low-dimensional representation is a *fuzzy hash*. Fuzzy hashes are produced by algorithms that differ from traditional cryptographic hashes because they are intentionally designed to map similar input values to similar hashes. Pagani et al. [58] studied the limitations of conventional fuzzy/locality sensitive

hashes computed over raw executables, concluding that small variations in the raw bytes of the input can significantly affect the generated hash. However, even if vanilla fuzzy hashes may not be suitable for function similarity, some approaches (like FunctionSimSearch [18]) have proposed more specialized hashing techniques to compare two functions.

Another popular form of low-dimensional representation relies on *embeddings*. The term, popular in the machine-learning community, refers to a low-dimensional space where semantically similar inputs are mapped to points that are close to each other, regardless of how different the inputs looked in their original representation. The goal of the machine-learning models is to learn how to produce embeddings that maximize the similarity among similar functions and minimize it for different functions. In the literature we can identify two main types of embeddings: those that try to summarize the code of each function and those that try to summarize their graph structure.

Code embeddings. Numerous researchers tried to adapt existing Natural Language Processing (NLP) techniques to tackle the binary function similarity problem by treating assembly code as text. These solutions process streams of tokens (e.g., instruction, mnemonic, operand, normalized instruction) and output one embedding per code block, one embedding per instruction, or both. A first class of approaches (e.g., Asm2Vec [14] and [64]) are based on word2vec [52, 53], a well-known technique in the NLP field. Although these models are not designed for cross-architecture embedding generation, they can be trained on different instruction sets at the same time, learning the syntax of different languages (but without being able to map the semantics across languages) or they can be applied on top of an intermediate language. A second line of solutions is based on seq2seq encoder-decoder models [69], which allows to map the semantics from different architectures to the same embedding space, thus learning cross-architecture similarity [49, 80, 82]. A third type of models builds on top of BERT [12], the state-of-the-art pre-training model in NLP based on the transformer [71]. For instance, OrderMatters [78] uses the BERT model pre-trained on four tasks to generate basic block embeddings, while Trex [60] uses a hierarchical transformer and the Masked-Language-Modeling task to learn approximate program execution semantics and then transfer the learned knowledge to identify semantically similar functions.

Assembly code embeddings are usually affected by the number of different instructions they can deal with (the so-called out-of-vocabulary problem (OOV)), and by the maximum number of instructions that can be provided as input to the model. As a result, certain approaches compute instruction-level embeddings [14, 16, 64], basic block embeddings [16, 78, 80, 82], or function-level embeddings [14, 49, 60]. Instruction or basic block embeddings are sometimes leveraged to compute function similarity by using other algorithms such as Longest Common Subsequence [82], or they are used as part of more complex models as detailed in the following category.

Graph embeddings. Another line of research builds on machine-learning approaches that compute embeddings for graphs. These are very suitable to capture features based on the function control-flow graphs, which are cross-architecture by nature. These embeddings can be generated by custom algorithms [24, 44] or by more complex machine-learning techniques, such as Graph Neural Network (GNN) [25, 40, 45, 76, 78, 79]. Some recent approaches from the machine-learning community propose variations of GNN, such as the GMN. These variations are able to produce embeddings comparable in a vector space [40, 43], with the particularity that these embeddings encode information from the two graphs provided as input to the model.

Graph embedding approaches also often encode information from each basic block in their corresponding node of the graph to add more expressiveness. For instance, some solutions compute a set of attributes for each node, thus leading to Attributed Control-Flow Graphs (ACFG), which can either be manually engineered [24, 76] or automatically learned in an unsupervised way [45]. Other authors leverage other embedding computation layers using some of the techniques discussed earlier (e.g., at basic block level [45, 78, 79]).

2.2 Function Representations

Binary functions are essentially streams of bytes corresponding to architecture-specific machine code and data. Starting from this raw input, researchers have used a number of ways to extract higher-level information that could be used to tell whether two functions originate from the same source code. The list, ordered by increasing level of abstraction, includes the following pieces of information.

Raw bytes. Some solutions directly use the raw binary information as a starting point for a similarity measure (e.g., Catalog1 [74]) or combine raw bytes with other information obtained from the control-flow graph (CFG) or the call graph (CG) [44].

Assembly. Assembly instructions, as obtained by a disassembler, can be useful when operations can be encoded in many different ways depending on the instruction size or its operands (e.g., in x86/64 architecture, a `mov` instruction can be encoded by using a number of different opcode bytes [33]). Approaches such as Asm2Vec [14] and Trex [60] benefit from this level of abstraction by using disassembled instructions as input, while others compute additional metrics such as "the number of arithmetic assembly instructions in a given function" [24, 25, 76].

Normalized assembly. Assembly code often encodes constant values (e.g., immediate operands and absolute or relative addresses), which result in a very high number of potential combinations of operations and operands. Assembly normalization is used in [22, 45, 49, 64, 80, 82] to abstract away some of this variability, reduce the vocabulary size, and

converge all the possible variations of the same operation into a single representation.

Intermediate representations. Some approaches work on an even higher abstraction level by lifting the binary representation to an intermediate representation (IR). The use of an IR brings several advantages: (i) it can unify the representation of semantically equivalent but syntactically different instructions, (ii) it potentially abstracts away non-relevant artifacts of different architectures, and (iii) it allows to apply program analysis techniques to simplify (and converge) certain code constructs. Existing works have employed a number of different IRs, such as LLVM [10, 23, 25], VEX [6, 10, 30, 67], and IDA microcode [78, 79].

Structure. Numerous approaches try to capture the internal structure of a given function, or the role that a function plays within the overall program. To capture a function's internal structure, many approaches [3, 18, 32, 56] extract the (intra-procedural) Control-Flow Graph (CFG). Some enrich the CFG with data obtained from the basic blocks, i.e., Attributed Control-Flow Graph (ACFG) [18, 20, 24, 25, 40, 51, 76, 78, 79, 81], or other types of graphs or information obtained from the function (e.g., register flow graph [1]) or its context within the binary (call graph [44, 68]). Finally, some techniques just benefit from the structure provided by the CFG to compute alternative features — such as tracelets (sequences of consecutive basic blocks in the CFG [11, 56]).

Data flow analysis. The implementation of an arithmetic expression at the assembly level may employ different forms to implement the same semantics. To deal with these scenarios, previous works proposed to first compute program slices based on data-flow dependencies, and to then normalize and use them as features to capture a function's behavior [9, 67]. Other papers, such as Vulseeker [25], employ data flow edges between blocks as an additional feature.

Dynamic analysis. Some approaches rely on dynamic analysis [19], e.g., by executing pairs of functions and extracting features from the relationship between the inputs and outputs [30, 62]. Other approaches simply extract semantic features derived from full or partial execution traces [29, 34, 39, 54, 72], while other leverage emulation [77] or hybrid [31, 60] techniques.

Symbolic execution and analysis. As opposed to concrete dynamic execution, some approaches rely on symbolic execution to fully capture the behavior of the function under analysis and to determine the relationship between its inputs and its outputs, under all possible paths [6, 46, 55].

3 Selected Approaches

One of the main contributions of our work is to provide a reference implementation for a number of key approaches and to compare them by performing experiments on a common and comprehensive dataset. Ideally, one would evaluate as many approaches as possible, but clearly it is not feasible to re-implement them all. It is also important to understand that, while there are hundreds of papers on the topic, many of them present small variations of the same techniques and the number of novel solutions is significantly lower.

In this section we first discuss our selection criteria, and we then introduce the set of techniques we implemented and evaluated.

3.1 Selection Criteria

Scalability and real-world applicability. We are interested in approaches that have the potential to scale to large datasets and that can be applicable to real-world use cases. Thus, we do not evaluate approaches that are inherently slow and only focus on direct comparisons, such as the ones based on dynamic analysis, symbolic execution, or high-complexity graph-related algorithms.

Focus on representative approaches and not on specific papers. There are many research works that propose just small variations of the same approach — for example by reusing previous techniques while slightly changing which features are used. This often results in a similar overall accuracy, which makes them less interesting for our comparison.

Cover different communities. The research contributions on the problem of binary function similarity come from different research communities and from both academia and industry. Unfortunately, it is often the case that research papers in a given community are only evaluated against proposals from the same community or, at times, only against previous works from the same authors. Thus, for our evaluation, we wanted to include representative research from the systems security, the programming language analysis, and the machine-learning communities. For completeness, we also considered approaches proposed by industry as well.

Prioritize latest trends. While the first contributions in this research field date back to more than a decade ago, there has been a recent surge in interest. Moreover, the majority of these recent publications employ, in one way or another, techniques based on machine learning. These techniques, in turn, have been reported to outperform all previous approaches. Some researchers have suggested that basic approaches work as well as machine-learning techniques, but our evaluation shows that this is the case only when considering simple evaluation scenarios. Thus, while we do consider various types of approaches, we do prioritize these latest, more promising, research trends.

3.2 Selected Approaches

In Section 2 we have presented the types of input data that researchers have extracted over the years as well as the

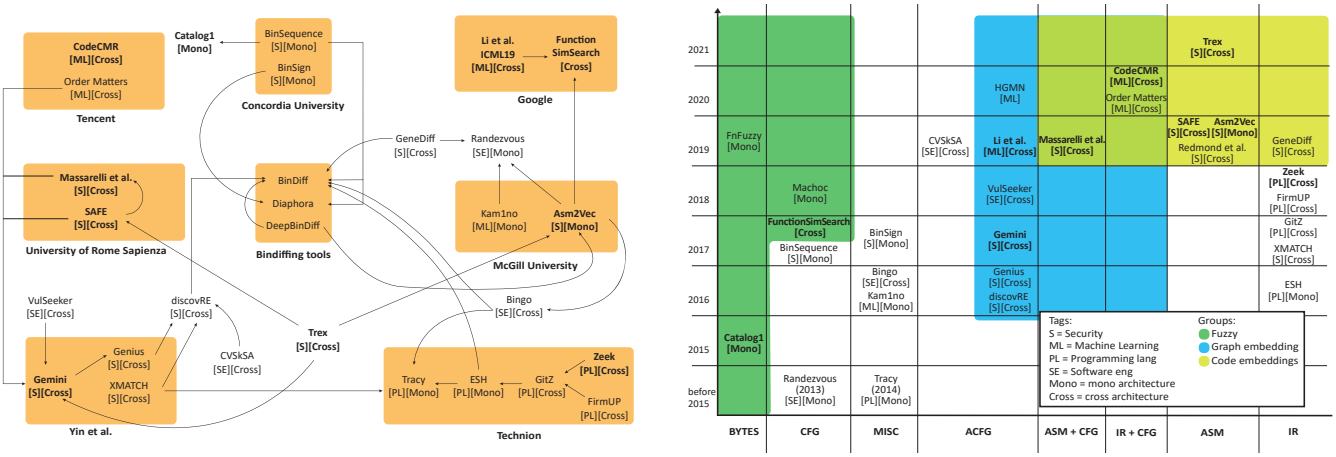

Figure 1: Function Similarity Systematization

possible methods to compute function similarity. However, only a subset of the many papers published over the last decade meet the criteria described above. Based on our analysis, we identified 30 techniques, represented in Figure 1, out of which we then selected *ten* representative solutions for our study.

The graph on the left of Figure 1 displays the approaches clustered according to their respective research group. These groups come from both academia and industry — with both Google and Tencent being very active in this area. The edges represent the other solutions to which each paper compares its results with. For instance, the arrow between Gemini and Genius means that the results of Gemini were compared by the authors with the results previously obtained by Genius (both from the same group). The right portion of Figure 1 shows instead the timeline of publication on the Y axis, and the different types of input data on the X axis. The approaches are then clustered in three main groups based on the different ways of computing the similarity, i.e., fuzzy hashes, graph embeddings, and code embeddings.

Both figures make use of tags (in brackets) to identify the community ([S] security, [PL] programming languages, [ML] machine learning, and [SE] software engineering). We also use the [Mono] and [Cross] tags to represent whether the proposed approaches focus on, respectively, mono- or cross-architecture scenarios.

Even if the graph in Figure 1 is not comprehensive and only shows the papers we selected, it depicts once again how several papers compare only against a limited set of previous approaches. There are also other interesting messages we can extract from these plots. First, the binary diffing tools grouped in the middle box [13, 16, 83] have all been designed for a direct comparison of two binaries (e.g., they use the call graph) and they are all mono-architecture. Nevertheless, several papers that proposed cross-architecture and function similarity solutions compare their results against these tools. This is clearly an issue that can lead to wrong conclusions and shows some flaws in the experiments and inappropriate evaluation strategies.

Second, the graph shows that the different communities are often quite hermetic and they rarely compare with papers from other fields. This is a clear limitation for advancing function similarity research and we hope this paper can foster collaboration among the different fields. Last, we can identify seminal papers such as Gemini [76] and discovRE [20] that have been re-implemented and tested extensively in other studies. These works have clearly inspired other researchers to improve the state of the art.

The timeline picture on the right shows a clear trend: the complexity of the solutions and the use of machine learning grew over time. We used this information and the relationships depicted in the picture to select *ten* state-of-the-art solutions that are scalable, representative, and recent. At the same time, we tried to maximize the variance between the research communities.

For instance, we selected Gemini [76] but not Genius and discovRE because Gemini outperformed Genius [24] in all its experiments and Genius outperformed discovRE [20]. Therefore, we are confident that Gemini also outperforms discovRE. Based on similar considerations we selected Asm2Vec [14], which demonstrated a better accuracy than Bingo [6]. We also retained some works that at first sight may seem outdated (e.g., Catalog1 [74]) because we believe they are representative to study the evolution of this field, have interesting results, and reflect our selection criteria. Finally, in some cases we may have selected more than one work in a given category. This can happen when there are concurrent works and it is not clear from the literature which one performs better. In the remaining part of this section we briefly describe the ten selected solutions.

Bytes fuzzy hashing: Catalog1. Catalog1 [74] is a fuzzy hashing approach based on the MinHash Locality Sensitive Hashing [4]. The algorithm takes as input the function bytes and produces a fixed length signature, which is a promising way to compare functions from the same architecture.

This work is from a non-academic community, and it is implemented in an IDA plugin.

CFG fuzzy hashing: FunctionSimSearch. FunctionSimSearch [18] uses the SimHash algorithm [7] to compute a fuzzy hash that combines graphlets (i.e., small connected, non-isomorphic, induced subgraphs) extracted from the CFG, mnemonics, and immediate values from the assembly code. The approach is potentially cross-architecture because of the CFG-based features. This tool is developed by a researcher from the industry.

Attributed CFG and GNN: Gemini. Gemini [76] uses a GNN (Structure2vec [8]) to compute a function embedding starting from the function ACFG (i.e., a control-flow graph with basic-block level attributes). This approach marks a milestone, because it is the first to leverage GNN with a Siamese architecture [5] to learn function similarity. This is clearly an evolution compared to basic ACFG-based solutions (Genius [24]) and it is more efficient than other approaches that leverage CFG data such as Bingo [6], Binsign [56], Kam1no [15], or Tracy [11].

Attributed CFG, GNN, and GMN: Li et al. 2019. The approach presented in [40] is proposed by researchers from the industry (DeepMind and Google) from the machine-learning community and it presents a novel graph matching model to compute the similarity between pairs of graphs. The authors explored function similarity as one of the practical use cases. This approach proposes two cutting-edge models from the machine-learning community that had not yet been studied by system security researchers. Moreover, the paper shows promising results.

IR, data flow analysis and neural network: Zeek. Zeek [67] performs dataflow analysis (slicing) on the lifted code (VEX IR) at the basic-block level and computes strands. Then, a two-layer fully-connected neural network is trained to learn the cross-architecture similarity task. This approach is the most advanced proposal combining intermediate representations, data flow analysis, and machine learning. This work outperforms previous research from the same authors.

Assembly code embedding: Asm2Vec. The Asm2Vec [14] NLP model derives from the PV-DM variant of paragraph2vec [38], an extension of the original word2vec [53] model. Asm2Vec introduces a finer instruction-level splitting and embedding construction in order to overcome the limitations of the out-of-vocabulary (OOV) problem with assembly instructions. This approach is fully unsupervised, and achieves state-of-the-art results in the mono-architecture experiments.

Assembly code embedding and self-attentive encoder: SAFE. SAFE [49] uses the self-attentive sentence encoder from Lin et al. [42] to learn cross-architecture function embeddings. This approach is representative of the NLP encoders from the seq2seq model, and, in contrast to Asm2Vec, it was specifically designed to learn cross-architecture similarity.

Assembly code embedding, CFG and GNN: Massarelli et al., 2019. Massarelli et al. [45] uses the same Structure2vec GNN of Gemini [76], but it changes the block-level features, switching from manually engineered features to unsupervised ones. This approach is interesting because it is an evolution of Gemini and it combines the advantages of instruction-level embeddings, basic-block encoder, and GNN.

CodeCMR/BinaryAI. The model presented in [79] powers the BinaryAI framework [70] for the binary source code matching at function-level. We only focus on the part that handles the function in binary format. The model combines intermediate representation with an NLP encoder to get basic-block embeddings and a GNN to obtain the graph embedding. Two LSTMs encode strings and integer data from the function. The function embedding is the concatenation of the three, and the binary model is trained end-to-end. We note that this work from Tencent is a follow-up of an authors' previous work [78] and the authors, when contacted, explained how the new model is more accurate.

Trex. Trex [60] is a recent work based on a hierarchical transformer and micro-traces. This paper brings a dynamic component that extracts function traces and that is fundamental to learn the semantics of the functions. The authors pretrain the ML model on these traces and transfer the learned knowledge to match semantically similar functions. The matching phase is based exclusively on static features while the emulation to generate the micro-traces is required only during the pretraining. This cross-architecture solution is built on top of the transformer, the state-of-the-art deep learning model in NLP.

4 Evaluation

4.1 Implementation

One of the goals of this study is to perform a fair comparison among the different approaches. For this reason, we implemented each phase of the evaluation in an uniform way, including the binary analysis, the feature extraction, and the machine-learning implementations. In this way, it is possible to create a common ground to perform a meaningful and fair comparison of the different methodologies.

For the binary analysis phase we used IDA Pro 7.3 [28], while for the feature extraction we relied on a set of Python scripts using the IDA Pro APIs, Capstone [63], and NetworkX [26]. We implemented all the neural network models in Tensorflow 1.14, with the only exception of Trex [60], which was built on top of Fairseq [57], a sequence modeling toolkit for PyTorch. Finally, we used Gensim 3.8 [65] to implement Asm2Vec [14] and to run the instruction embedding models [45, 49].

In several cases we were able to obtain at least a portion of the original code base of the underlying research works [17, 21, 48, 50, 59, 74, 75]. Unfortunately, even when part of the code

was available, it was often tailored to the dataset the authors used in their paper, and we had to put a substantial implementation effort to make it execute correctly on a different set of test cases. During this process, we adopted a uniform implementation to minimize evaluation differences and we introduced several code optimizations. When the code was not available, we contacted the authors, but we received either no answer or limited support. Two approaches, Zeek [67] and Asm2Vec [14], have been fully reimplemented, while CodeCMR was tested by the authors due to the high complexity of the model and several "hidden" variables not discussed in the paper.

Additional technical details of all our implementations, together with information regarding our effort to contact the respective authors and the considerations regarding the use of pre-trained models, are available in [47].

We run all the experiments on a workstation equipped with Ubuntu 18.04, Intel Xeon Gold 6230 (80 virtual cores @2.10 GHz), 128GB DDR4 RAM, and one Nvidia RTX2080 Ti GPU (1350MHz, 11GB GDDR6, 13.45 TFLOPS FP32).

4.2 Dataset

We created two new datasets, Dataset-1 and Dataset-2, which aim at capturing the complexity and variability of real-world software, while covering the different challenges of binary function similarity: (i) multiple compiler families and versions, (ii) multiple compiler optimizations, (iii) multiple architectures and bitnesses, and (iv) software of different nature (command line utilities vs. GUI applications). We use Dataset-1 to train the machine-learning models and both datasets to test the evaluated approaches.

Dataset-1. Dataset-1 is composed of seven popular open-source projects: `ClamAV`, `Curl`, `Nmap`, `OpenSSL`, `Unrar`, `Z3`, and `Zlib`. Once compiled, they produce 24 distinct libraries. Each library is compiled using two compiler families, `GCC` and `Clang`, with four different versions each, covering major releases from 2015 to 2021 (additional details on the open-source projects and compiler versions are provided in [47].) Each library is compiled for three different architectures, x86-64, ARM, and MIPS, in 32 and 64 bit versions (with a total of 6 architecture combinations), and 5 optimization levels `O0`, `O1`, `O2`, `O3`, and `Os`.

Following our definition of function similarity, we disabled function inlining to compare functions originating from exactly the same source code: function inlining is in fact an addition of code into the original source code, and it could potentially pollute our results and lead to misleading conclusions.

In total, this dataset consists of 5,489 binaries, with an average of 228 combinations per binary project, and a total of 26.8M functions. Following the criteria applied by several seminal papers [20, 24], we filtered out the functions that had less than five basic blocks (18.2M). 80% of the filtered functions correspond to functions with two basic blocks, and 93% of these have less than 30 assembly instructions. The remaining 8.6M functions are the starting point for the construction of the training, validation and testing dataset. The Appendix includes additional information about the number of basic blocks and instructions for the selected functions.

Dataset-2. Dataset-2 is built on top of the binaries released by the authors of Trex [60], a very recent paper. In particular, we selected 10 libraries out of 13 to avoid any intersection with Dataset-1: `Binutils`, `Coreutils`, `Diffutils`, `Findutils`, `GMP`, `ImageMagick`, `Libmicrohttpd`, `LibTomCrypt`, `PuTTy`, and `SQLite`. The dataset contains binaries *already* compiled for x86, x64, ARM 32 bit and MIPS 32 bit, 4 optimization levels (`O0`, `O1`, `O2`, `O3`) and `GCC-7.5`. Dataset-2 complements our Dataset-1 with the purpose of i) validating the results of the models of Dataset-1 on a diverse and large collection of binaries, and ii) including the comparison with the recent Trex approach. Indeed, Trex could not be pre-trained and fine-tuned on Dataset-1, because the emulator released by the authors [59] only supports a subset of the architecture and bitness combinations. Moreover, as detailed in our online material in [47], Trex is extremely expensive to retrain, and we opted to use the same model as the authors in their experiments on the same set of binaries. We acknowledge the potential advantage of Trex in this evaluation compared to the other models trained on Dataset-1 (thus on a larger set of architectures), and we keep this into consideration in our discussion.

Dataset availability. To the benefit of the community and to ease future works in the area, we are releasing the full dataset to the public, available in [47]. We also release the scripts and patches we used to compile them so that future researchers can re-create the dataset and build on top of our work.

4.3 Experimental Settings

We performed extensive experiments to evaluate the accuracy of the selected approaches and several of their variants. To this end, we identify *six* different tasks to evaluate: (1) XO: the function pairs have different optimizations, but the same compiler, compiler version, and architecture. (2) XC: the function pairs have different compiler, compiler versions, and optimizations, but same architecture and bitness. (3) XC+XB: the function pairs have different compiler, compiler versions, optimizations, and bitness, but same architecture. (4) XA: the function pairs have different architectures and bitness, but the same compiler, compiler version, and optimizations. (5) XA+XO: the function pairs have different architectures, bitness, and optimizations, but the same compiler and compiler version. (6) XM: the function pairs come from arbitrary architectures, bitness, compiler, compiler versions, and optimization.

The first three tasks evaluate the techniques for those use cases that are limited to a single architecture only, including some practical applications of function similarity in malware analysis and collaborative reverse engineering. The fourth task is relevant for the analysis of firmware images cross-compiled

by using always the same compiler and compiler options. The fifth task is designed to support Dataset-2, which is compiled with only one compiler and compiler version. Finally, the last task is the most challenging and includes comparisons across the entire dataset. In our evaluation we also consider three sub-datasets for XM: XM-S, XM-M, and XM-L, which include small-sized functions (with less than 20 basic blocks), medium (between 20 and 100), and large ones (more than 100 blocks).

Each task is evaluated according to two different tests: (i) the area under curve (AUC) of the receiver operating characteristic (ROC) curve, which is an aggregate measure of the performance of a model across all the possible classification thresholds, and (ii) two commonly used ranking metrics, the mean reciprocal rank (MRR), and the recall (Recall@K) at different K thresholds. Ranking measures are useful to evaluate the model performances in those applications where it is necessary to search candidates functions through a large database, such as in the vulnerability research use case.

For the first test, we constructed a dataset of 50k positive pairs and 50k negative pairs for each task, with a total of 700k function pairs across Dataset-1 and Dataset-2. For the ranking test, we selected 1,400 positive pairs and 140k negative pairs, that is 100 negative pairs for each positive one.

Overall our tests cover 438,981 unique binary functions, with the constraint of having at least five basic blocks. In every task, the pairs are randomly sampled according to the corresponding constraints (e.g., in cross-optimization, positive and negative pairs belong to the same architecture).

To make our evaluation more meaningful, we opted to remove duplicate functions (by checking for their names and the hash of their instructions). We also note that this "pair selection" aspect is crucial for a proper evaluation. Unfortunately, several recent works use different ways to select such pairs, making their training tasks significantly easier or more difficult. For example, in [36], the researchers generated negative pairs while keeping the same compiler options. In another example (VulSeeker [25]) the negative pairs are randomly generated and they do not enforce any check on the compiler options as well as the presence of the anchor function. These inconsistencies hinder the potential for fair comparisons, and they are often overlooked.

By following the common practice in this research area, we present the evaluation of the selected approaches in the most generic way, using two specifically designed datasets. We note that directly evaluating the approaches on a security application is usually challenging, especially in the case of vulnerability discovery and malware clustering, because it requires to create an accurate ground-truth, which is the problem that function similarity is trying to solve in the first place.

4.4 Fuzzy-hashing Comparison

This section discusses our in-depth investigation of the two approaches based on fuzzy hashing: Catalog1 [74] and FunctionSimSearch [18].

Table 1: AUC Comparison of Catalog1 and FunctionSimSearch (FSS) when varying only one variable at the time: Architecture, Bitness, Compiler, Optimizations, Compiler Version

		Free variable				
	Description	Arch	Bit	Comp	Opt	Ver
Catalog1	B + size 16	0.49	0.63	0.63	0.75	0.94
Catalog1	B + size 128	0.43	0.76	**0.85**	**0.92**	**0.99**
FSS	G	**0.81**	**0.89**	0.68	0.74	0.87
FSS	G + M	0.66	0.88	0.78	0.83	0.97
FSS	G + M + I	0.67	0.88	0.77	0.82	0.97
FSS	w(G + M + I)	0.75	0.83	0.67	0.74	0.82

Catalog1 uses raw bytes as input features and a different signature size (i.e., number of hash functions): we show the results of two variants, one with size 16 and the other with size 128. In contrast, FunctionSimSearch (FSS) uses a combination of graphlets (G), mnemonics (M) and immediates (I): we did different tests by incrementally enabling different types of input features, including their weighted linear combination *w*, as we found it in the original implementation (G:1.00, I:4.00, M:0.05).

Since fuzzy hashing approaches are not influenced by a training phase, we used them to perform a targeted evaluation of how each compilation variable affects the comparison of binary functions. Thus, for these approaches, we first perform multiple experiments in which we vary one variable (i.e., compiler family, version, options, architecture, and bitness) while we keep the remaining ones the same. The results, in Table 1, make clear that, when considering only one free variable at a time, even simple approaches such as fuzzy hashes are effective: "raw" bytes are confirmed to be good features for same-architecture comparisons, while graphlets are effective in cross-architecture comparisons. For Catalog1, the bigger the signature size, the better the performances, but they are limited by the total number of hash functions included in the implementation.

We then evaluated these two approaches with the six tasks presented earlier. Tables 2 and 4 show the results on Dataset-1 and Dataset-2: having multiple free variables at the same time is a much harder problem and simple approaches are not effective anymore. In the XC task (Table 2), Catalog1 and FSS have identical AUC. For FSS, the graphlets-only (G) configuration is the best in all the tasks except XC and XO, where using graphlets with mnemonics (G+M) has higher AUC. Moreover, FSS also seems to work better on bigger functions, which may be due to the higher number of different graphlets that can be extracted. Finally, in the XA task, FSS accuracy decreases when using additional features such as mnemonics and immediates, and surprisingly, the weighted linear combination of the three features do not produce better results than other basic configurations. Catalog1 is the fastest among the two approaches, while FSS is about 3 times slower due to a longer feature extraction phase.

Table 2: Comparison of Catalog1 and FunctionSimSearch (FSS) on Dataset-1.

						XM					XC+XB	
	Description	XC	XC+XB	XA	XM	small	medium	large	MRR10	Recall@1	MRR10	Recall@1
Catalog1	B + size 16	0.66	0.60	0.48	0.54	0.54	0.53	0.54	0.08	0.07	0.25	0.23
Catalog1	B + size 128	**0.73**	0.66	0.43	0.55	0.54	0.55	0.58	0.12	0.09	**0.31**	**0.27**
FSS	G	0.72	**0.72**	**0.69**	**0.70**	**0.70**	**0.71**	**0.77**	**0.26**	**0.20**	0.29	0.23
FSS	G + M	**0.73**	0.71	0.58	0.65	0.64	0.66	0.70	0.17	0.13	0.29	0.24
FSS	G + M + I	**0.73**	0.70	0.58	0.65	0.64	0.66	0.71	0.15	0.09	0.28	0.23
FSS	*w* (G + M + I)	0.69	0.69	0.65	0.67	0.66	0.68	0.72	0.21	0.16	0.23	0.17

4.5 Machine-learning Models Comparison

We evaluate all selected approaches by using a common training dataset extracted from Dataset-1 (with the exception of Trex [60]) and using a similar criteria to the XM task to create positive and negative samples. However, it is important to note that it would be possible to further improve the results of each task by using task-specific training data. We did perform this evaluation but we omit the results as we noticed that training on the most generic data (XM) achieves overall performances that are close to the best for each task.

Comparing machine-learning models, especially deep neural networks, is a challenging task because several variables may influence the final results, including the model implementation and configuration (e.g., number of layers or the type of recurrent neural network), the different hyperparameters (e.g., the learning rate and the batch size), the loss function, the optimizer, and the number of training epochs. To be as uniform as possible in our comparison, all models were trained with the same randomly generated data extracted from 256,625 unique binary functions. Moreover, we performed extensive experiments to evaluate different sets of features, different model configurations, hyperparameters, and loss functions. The results for each model can be improved by using an extensive grid search approach and the results we present can be used as a starting point for future works.

Table 3 and Table 4 show the results of the tested models and their respective variants on the two datasets. Table 8 includes some generic information about the models and their training, such as the number of parameters, the batch size, the number of epochs, and the training time for each epoch.

Results show that among the models that produce a vector representation of a function (i.e., an embedding) *the GNN from [40] achieves the best values in all the metrics and in all the tasks*. We also note how most of the machine-learning models perform quite similarly when compared on AUC, but differently on ranking metrics (MRR10 and recall@1), as shown in Figure 2. Then, regarding the other embedding models, SAFE [49] provides better AUC than GNN with unsupervised features [45], and in one specific configuration slightly better AUC than Gemini [76]. For the approaches that perform direct comparisons, *the GMN from [40] is the best performing model in all the tasks*, while Zeek has a slightly lower AUC (except for large functions), but much lower MRR10 and recall@1.

We now discuss in depth eight observations from our results.

Comparing Trex. Our results show that, on the XO task, Trex has the same AUC and similar MRR10 and recall@1 as Asm2Vec, as well as the other word2vec variants, which is slightly lower than the one of GNN and GMN. This consolidates the finding that language models are strong in the same architecture tasks. However, differently from Asm2Vec, Trex retains the same performances also in the XA and XA+XO task, which shows the power of the transformer in (cross language) cross-architecture setup. We confirm that Trex improves SAFE performances in the XO task (0.94 vs. 0.90) and in the XA+XO task (0.94 vs. 0.91). Our tests also show that the inference for Trex running on GPU (more details in [47]) is faster than our multiprocess Asm2Vec implementation (3.92s vs. 8.51s for 100 functions), but it is slower than SAFE running on CPU (3.92s vs. 1.46 for 100 functions).

Comparing different GNNs. Both Gemini [76] and Li et al. [40] use GNNs with a different variant of the message passing and aggregation layer [8, 41]. Using the results from the third and fourth block of lines in Table 3 and Table 4, we compare the two variants using both basic block features (the bag of words (BoW) of the opcodes) and no features. Results show that the GNN variant of Li et al. provides a significant improvement over GNN (s2v), the one used by Gemini, in all the tasks. However, the execution time of two variants remain similar (1.48s vs. 1.40s when using no features).

Comparing different feature set in GNN (s2v). Gemini [76] uses a GNN model called Structure2vec (s2v) [8] with manually engineered features. Our goal is to understand how important these features are with respect to not having a feature vector at all or using another set of features such as the bag of words (BoW) of the opcodes. Results in Table 3 and Table 4 show that manually engineered features only perform better on small and large functions in the XA task, and that the BoW of the opcodes perform similarly in all the different metrics, and even have better recall for different K values, as shown in Figure 2. Also, the execution time is different (1.66s vs. 7.18s), due to a longer feature extraction phase in Gemini. This means that more complex and difficult to extract features do not necessarily outperform a more basic representation. A BoW of 200 opcodes has 20 times the number of features of Gemini, which results in a larger input matrix for the node neural network of GNN. We additionally

Table 3: Comparison of machine-learning models on Dataset-1.

	Description	XC	XC+XB	XA	XM	XM small	XM medium	XM large	XM MRR10	XM Recall@1
[67] Zeek (direct comparison)	Strands	0.84	0.85	0.84	0.84	0.85	0.83	**0.87**	0.28	0.13
[40] GMN (direct comparison)	CFG + BoW opc 200	0.85	0.86	**0.86**	0.86	**0.89**	0.82	0.79	**0.53**	**0.45**
[40] GMN (direct comparison)	CFG + No features	**0.86**	**0.87**	**0.86**	**0.87**	0.88	**0.85**	0.84	0.43	0.33
[40] GNN	CFG + BoW opc 200	**0.86**	**0.87**	**0.86**	**0.87**	**0.89**	0.84	0.76	0.52	0.44
[40] GNN	CFG + No features	0.82	0.83	0.82	0.82	0.85	0.80	0.76	0.37	0.29
[76] GNN (s2v)	CFG + BoW opc 200	0.81	0.82	0.78	0.81	0.82	0.78	0.74	0.36	0.26
[76] GNN (s2v)	CFG + manual	0.81	0.82	0.80	0.81	0.84	0.77	0.79	0.36	0.28
[76] GNN (s2v)	CFG + No features	0.69	0.70	0.69	0.70	0.70	0.69	0.75	0.12	0.07
[45] w2v + AVG + GNN (s2v)	CFG + N. asm 150	0.79	0.79	0.74	0.77	0.78	0.75	0.73	0.24	0.16
[45] w2v + wAVG + GNN (s2v)	CFG + N. asm 150	0.79	0.79	0.76	0.77	0.78	0.76	0.76	0.29	0.20
[45] w2v + RNN + GNN (s2v)	CFG + N. asm 150	0.79	0.80	0.79	0.80	0.82	0.77	0.80	0.27	0.17
[49] w2v + SAFE	N. asm 150	0.80	0.81	0.80	0.81	0.83	0.77	0.77	0.17	0.07
[49] w2v + SAFE	N. asm 250	0.82	0.83	0.82	0.83	0.84	0.81	0.82	0.22	0.09
[49] w2v + SAFE + trainable	N. asm 150	0.80	0.81	0.80	0.81	0.83	0.76	0.74	0.29	0.16
[49] rand + SAFE + trainable	N. asm 150	0.79	0.80	0.79	0.80	0.83	0.75	0.74	0.28	0.17
[14] Asm2Vec	10 CFG random walks	0.77	0.69	0.60	0.65	0.63	0.70	0.78	0.12	0.07
[38] PV-DM	10 CFG random walks	0.77	0.70	0.50	0.62	0.63	0.62	0.61	0.11	0.08
[38] PV-DBOW	10 CFG random walks	0.78	0.70	0.50	0.63	0.63	0.62	0.61	0.11	0.09

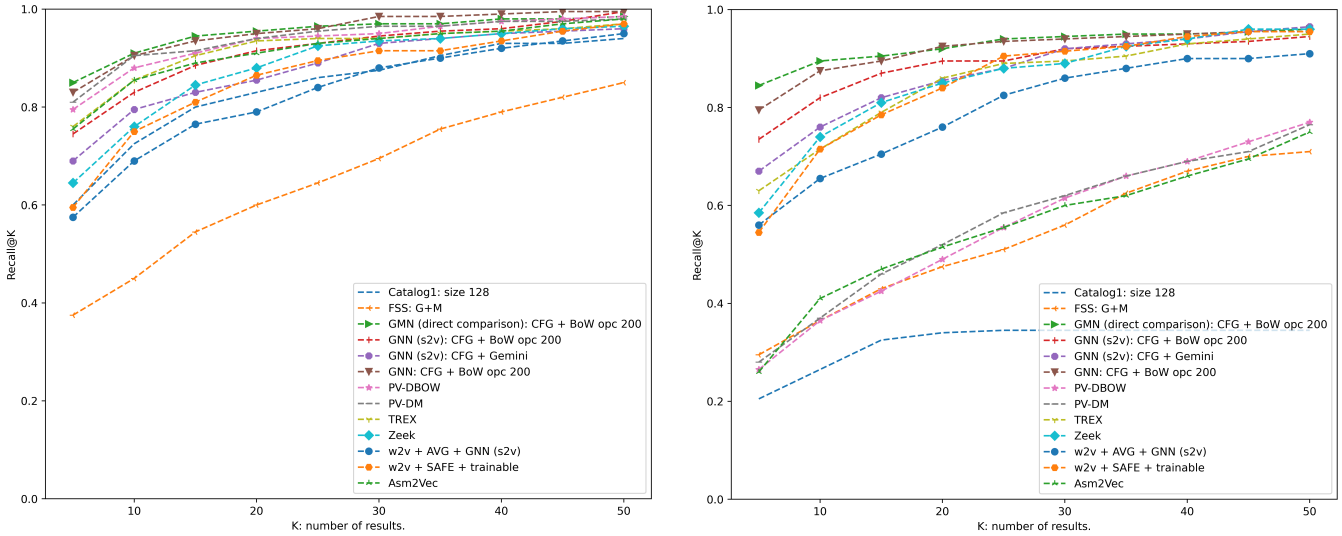

Figure 2: Comparison of the recall at different K values for XO (left) and XM (right) tasks.

tested a BoW of 1024 opcodes but the results did not improve significantly, which means that those additional features do not contribute significantly to the representation of the function.

Finally, we wanted to test whether the use of instructions embeddings as GNN features helps to increase the AUC, as presented in [45]. Our results show that the instruction embeddings on the normalized assembly do not have higher AUC than bag of words of the opcodes or manual engineered features (only the RNN basic block encoder achieves similar AUC), MRR10 and the recall@1 are lower too and the training time is drastically increased.

Modelling functions using a textual encoder. SAFE [49] uses a sentence encoder based on instruction embeddings, and the AUC is better than GNN (s2v) with unsupervised features [45]. Compared to Gemini [76], the AUC is similar, but MRR10 and recall@1 are lower. SAFE works better on small functions and results do improve when increasing the maximum instruction length from 150 to 250 (Table 3). However, SAFE needs to confront the challenge of out-of-vocabulary (OOV) words. As introduced in Section 2, other approaches such as InnerEye [82] or Mirror [80] apply different assembly normalization techniques to mitigate this issue. To illustrate this challenge, we measured the impact of OOV instructions in SAFE and we observed that x86-64 is the most affected architecture by the OOV problem (less than

Table 4: Comparison of fuzzy hashing and machine-learning models on Dataset-2

Model name	Description	AUC			MRR10			Recall@1			Testing time (s)		
		XO	XA	XA+XO	XO	XA	XA+XO	XO	XA	XA+XO	Feat	Inf	Tot 100
[67] Zeek (direct comparison)	Strands	0.92	0.94	0.91	0.42	0.45	0.36	0.28	0.31	0.21	7225.41	67.00	9.92
[40] GMN (direct comparison)	CFG + BoW opc 200	**0.97**	**0.98**	**0.96**	**0.75**	**0.84**	**0.71**	**0.66**	**0.77**	**0.61**	1093.68	1005.00	1.83
[40] GMN (direct comparison)	CFG + No features	0.93	0.97	0.95	0.61	0.76	0.67	0.51	0.68	0.59	978.15	876.00	1.63
[40] GNN	CFG + BoW opc 200	0.95	0.97	0.95	0.67	0.79	0.67	0.57	0.73	0.57	1093.68	116.52	1.66
[40] GNN	CFG + No features	0.91	0.96	0.93	0.54	0.71	0.59	0.44	0.62	0.49	978.15	100.34	1.48
[76] GNN (s2v)	CFG + BoW opc 200	0.94	0.95	0.93	0.58	0.57	0.58	0.48	0.42	0.47	1093.68	118.59	1.66
[76] GNN (s2v)	CFG + Gemini	0.93	0.96	0.93	0.57	0.74	0.57	0.47	0.64	0.49	5139.91	98.40	7.18
[76] GNN (s2v)	CFG + No features	0.75	0.79	0.77	0.18	0.20	0.23	0.12	0.13	0.16	978.15	40.87	1.40
[45] w2v + AVG + GNN (s2v)	CFG + N. asm 150	0.90	0.88	0.87	0.46	0.31	0.42	0.38	0.18	0.33	1070.01	258.95	1.82
[45] w2v + wAVG + GNN (s2v)	CFG + N. asm 150	0.87	0.87	0.85	0.37	0.29	0.36	0.29	0.17	0.27	1070.01	253.72	1.81
[45] w2v + RNN + GNN (s2v)	CFG + N. asm 150	0.88	0.90	0.88	0.32	0.35	0.35	0.19	0.18	0.23	1070.01	685.50	2.41
[49] w2v + SAFE	N. asm 150	0.88	0.90	0.88	0.27	0.30	0.31	0.14	0.18	0.20	1031.23	33.33	1.46
[49] w2v + SAFE	N. asm 250	0.86	0.88	0.87	0.28	0.32	0.28	0.16	0.19	0.19	1031.23	33.33	1.46
[49] w2v + SAFE + trainable	N. asm 150	0.91	0.93	0.91	0.40	0.43	0.37	0.26	0.25	0.23	1031.23	33.57	1.46
[49] rand + SAFE + trainable	N. asm 150	0.90	0.91	0.90	0.28	0.33	0.31	0.14	0.17	0.21	1031.23	33.81	1.46
[14] Asm2Vec	Rand walks asm	0.94	0.69	0.75	0.60	0.07	0.22	0.49	0.02	0.18	978.15	5235.00	8.51
[38] PV-DM	Rand walks asm	0.94	0.66	0.72	0.64	0.08	0.23	0.51	0.05	0.19	978.15	5239.00	8.52
[38] PV-DBOW	Rand walks asm	0.94	0.66	0.72	0.63	0.07	0.23	0.50	0.03	0.20	978.15	3004.00	5.46
[60] Trex	512 Tokens	0.94	0.94	0.94	0.61	0.50	0.53	0.50	0.38	0.46	1493.58	1365.89	3.92
[74] Catalog_1	size 16	0.72	0.50	0.55	0.43	0.06	0.14	0.38	0.06	0.14	654.70	0.00	0.90
[74] Catalog_1	size 128	0.86	0.48	0.57	0.50	0.07	0.17	0.42	0.06	0.14	823.47	0.00	1.13
[18] FSS	G	0.77	0.81	0.77	0.26	0.35	0.32	0.18	0.26	0.26	1903.46	466.07	3.25
[18] FSS	G + M	0.79	0.68	0.69	0.29	0.15	0.21	0.23	0.09	0.15	1903.46	466.07	3.25
[18] FSS	G + M + I	0.80	0.68	0.69	0.30	0.16	0.20	0.23	0.10	0.14	1903.46	466.07	3.25
[18] FSS	w(G + M + I)	0.83	0.80	0.78	0.43	0.30	0.36	0.36	0.23	0.29	1903.46	466.07	3.25

30% of the functions have no OOV words), probably due to its CISC instruction set, followed by MIPS, and finally ARM, with more than 40% of functions without a single OOV word.

Asm2Vec and other paragraph2vec models. Table 3 and Table 4 show the comparative results of Asm2Vec [14] with the PV-DM and PV-DBOW variants of paragraph2vec [38]. All the three models perform similarly, and compared to GNN [40], using a specific mono-architecture approach does not bring any advantage. We note that results are strongly influenced by several factors, including the size of the vocabulary of instructions, the number of random walks, and several implementation details. During the training, we selected 1M tokens (out of 1.9M) with a minimum frequency of 5, where most of them are numerical offsets or hexadecimal addresses. Lowering that threshold does improve the results, but also increases the size of the vocabulary and the training time. At inference time we did not change the vocabulary, even though these unsupervised methods can benefit from a new inference vocabulary without invalidating the results. In our tests all the three models share the same vocabulary. We also note that all the three variants achieve high AUC on scenarios where only one variable is free, e.g., the XO task in Table 4, but the AUC drops when multiple compilation variables are considered together, e.g., the XC task in Table 3, where the compiler, its version, and the optimizations change.

Comparing efficiency. We also kept track of how efficient these approaches are for what concerns training (Table 8) and testing time, i.e., inference time (Table 4). We focus the discussion on the second, because training the models is mostly a one-time effort. Regarding the inference time, SAFE [49] appears to be the fastest among the machine-learning models with 1.46s to process 100 functions. GMN and GNN from [40] have similar and among the lowest running times, however GMN processes only a pair of functions in input. GNN (s2v) with Gemini features [76] is 4 times slower than the version with opcode features: the reason is the longer feature extraction time. Zeek [67] inference time is also affected by long feature extraction and processing time, and it is the slowest among the approaches. GNN (s2v) with unsupervised features is slower in the RNN variant, due to a longer inference time given by the additional model complexity. Similarly, Trex [60] is affected by a long inference time due to the complexity of the NLP model. Finally, Asm2Vec [14] is among the slowest, because it requires 10 epochs of inference to extract the new function embeddings. Interestingly, Asm2Vec is slower than the other paragraph2vec [38] models due to the particular instruction embedding construction.

CodeCMR/BinaryAI evaluation. For our evaluation we also included CodeCMR, a recent paper by Tencent [79]. There were several problems attempting to replicate this work and its predecessor [78] (more details in [47]), but the authors were kind enough to assist us with an evaluation on our dataset. We

Table 5: Comparison of CodeCMR/BinaryAI with GNN and bag of words (BoW) of opcodes (opc) or IDA microcode (IR).

						XM				
	Description	XC	XC+XB	XA	XM	small	medium	large	MRR10	Recall@1
[40] GNN	CFG + BoW opc 200	0.86	0.87	0.87	0.87	0.90	0.84	0.78	0.58	0.52
[40] GNN	CFG + BoW IR 80	0.87	0.87	0.87	0.88	0.89	0.86	0.81	0.62	0.56
[79] CodeCMR/BinaryAI	CFG + IR + Int + Strings	**0.98**	**0.98**	**0.98**	**0.98**	**0.99**	**0.97**	**0.93**	**0.86**	**0.83**

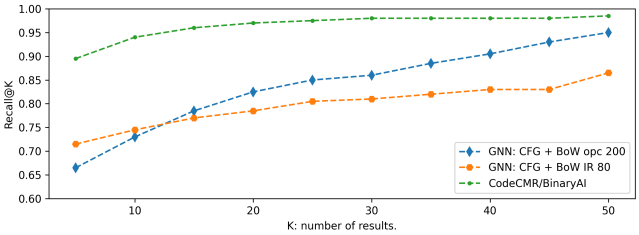

Figure 3: Comparison of the recall at different K values for the XM task for GNN [40] and CodeCMR/BinaryAI.

acknowledge that this kind of evaluation does not align with the re-implementation efforts we performed for the rest of the approaches. However, we believe CodeCMR to be quite interesting and promising, and we found it valuable to add it in our paper. We also note that one other option would have been to attempt to re-implement this approach, but we believe it would be extremely challenging to be confident about the faithfulness of our reimplementation, due to the high complexity of the system and many "hidden" variables not discussed in the paper. We now discuss insights from our evaluation.

Since the ultimate goal of their model is to match binary with source code, to train and test our data they isolated the part of the model that handles the function binary data only. We shared with them a pre-processed version of a subset of our dataset which consisted of ARM and x86 functions, 32 and 64 bit, that preprocessed using IDAPro with the HexRays decompiler.[2]

The extracted data consists of an attributed CFG with the IDA microcode instructions, integer number of ctree and the strings from the functions data. To have a baseline comparison, we run the GNN model from [40] using as basic block features the bag of words of the 200 most frequent opcodes and the bag of words of the 80 IDA microcode instructions.

Results are shown in Table 5. The GNN model that uses the BoW of the IDA microcode instructions has a higher AUC than the GNN model that uses BoW of the opcodes, but the second one has higher recall for large K values (Figure 3). In general, all the metrics for the BinaryAI/CodeCMR model are higher than the rest of our tested models. If these results are verified by independent studies in the community, this may be a very promising direction of research. In fact, the BinaryAI/CodeCMR model introduces several innovations. First, it merges in a single model several building components (i.e., a NLP encoder, a GNN, and two LSTMs), and it trains everything jointly by using an end-to-end strategy. Second, the authors show in their paper that the training strategy (e.g., using distance weighted sampling [73]) and loss function (e.g., triplet loss [66]) play an important role and can yield significant performance improvements.

Notes about Kim et al. In a recent paper posted on Arxiv by Kim et al. [36], the authors propose an interpretable model and show that manual feature engineering can achieve comparable results with "the state-of-the-art models," namely Vulseeker [25]. Their evaluation takes into account only one variable change at a time (e.g., only the compiler changes, while the architecture and the optimization level are fixed). This is a simplified setting compared to our six evaluation tasks, where, as shown in Table 1, even simple fuzzy-hashes approaches are effective. Moreover, the paper lacks any meaningful evaluation against state-of-the-art techniques. For example, the comparison with the ROC curve of Vulseeker is theoretical, performed on a different dataset, and without re-training the model. Another aspect that makes the comparison very challenging is how the positive and negative pairs are selected, as already mentioned in Section 4.3.

4.6 Vulnerability Discovery Use Case

As an example of a security application, we tested all models on a vulnerability discovery task. To this end, we selected ten vulnerable functions from OpenSSL1.0.2d, covering a total of eight CVEs. As a target, we selected the libcrypto libraries embedded in two firmware images: Netgear R7000 (ARM 32 bit) and TP-Link Deco M4 (MIPS 32 bit). Detailed information about which vulnerabilities affect each firmware image are included in [47]. We compiled the ten vulnerable functions for four architectures (x86, x64, ARM 32 bit, MIPS 32 bit) and we performed a ranking evaluation, similar to the one we have presented in the previous tests. When evaluating the vulnerability discovery results, we only used as a query the functions that are vulnerable for a particular firmware image. Results are shown in Table 7: we use the MRR10 as a comparison metric to evaluate how each model ranks the target vulnerable function for each query function. Unsurprisingly, the GMN model [40] with opcode features is the best performing one, however it requires analyzing each pair of functions,

[2] We could process only a subset of our dataset because we did not have access to a recent version of the HexRays MIPS decompiler license (BinaryAI plugin requires IDA 7.3 or later version). Moreover, HexRays only supports the decompilation of MIPS 32 bit code.

Table 6: Vulnerability test ranks.

		Netgear R7000			
Model name	Description	x86	x64	ARM32	MIPS32
[67] Zeek (direct comparison)	Strands	47;3;6;45	67;4;3;36	3;4;2;14	95;5;12;141
[40] GMN (direct comparison)	CFG + BoW opc 200	**1;1;1;2**	1;1;30;7	**1;1;1;1**	**1;1;1;7**
[40] GMN (direct comparison)	CFG + No features	4;1;1;3	5;1;52;2	2;1;1;4	7;1;1;12
[40] GNN	CFG + BoW opc 200	3;32;1;18	9;6;1;40	4;1;1;44	97;5;1;138
[40] GNN	CFG + No features	23;12;1;59	28;9;523;82	1;8;1;24	62;12;1;537
[76] GNN (s2v)	CFG + BoW opc 200	8;1;5;36	9;1;14;8	2;1;1;6	35;5;1;7
[76] GNN (s2v)	CFG + Gemini	432;3;1;155	372;6;66;60	356;2;1;77	339;15;1;308
[76] GNN (s2v)	CFG + No features	1295;6;10;160	1161;537;169;475	1149;228;9;690	1164;68;852;198
[45] w2v + AVG + GNN (s2v)	CFG + N. asm 150	37;76;5;5	3;35;45;39	30;1;1;70	76;74;54;100
[45] w2v + wAVG + GNN (s2v)	CFG + N. asm 150	1111;50;275;5	195;8;158;14	1123;13;54;7	438;25;661;14
[45] w2v + RNN + GNN (s2v)	CFG + N. asm 150	113;43;312;102	297;83;423;102	153;15;145;10	56;12;389;19
[49] w2v + SAFE	N. asm 150	65;164;146;99	62;46;147;39	106;41;85;40	93;29;133;54
[49] w2v + SAFE	N. asm 250	51;121;16;83	71;50;61;53	49;87;42;77	122;151;13;66
[49] w2v + SAFE + trainable	N. asm 150	5;77;9;28	7;1;92;27	19;1;15;6	184;15;2;7
[49] rand + SAFE + trainable	N. asm 150	55;161;105;15	24;51;175;30	134;64;50;36	319;180;54;10
[60] Trex	512 Tokens	41;4;1;3	10;3;1;2	32;4;1;2	24;16;1;1
[14] Asm2Vec	Rand walks asm	1;13;109;58	2;36;11;449	2;1;1;1	31;1;257;1
[38] PV-DM	Rand walks asm	15;7;9;1	13;13;395;454	119;1;1;1	2;71;272;2
[38] PV-DBOW	Rand walks asm	25;7;22;7	24;19;242;19	109;2;1;1	1;14;519;10
[74] Catalog_1	size 16	452;868;1156;947	452;868;1156;947	20;849;4;1082	452;868;1156;947
[74] Catalog_1	size 128	454;868;1163;947	453;868;1156;949	60;1;1;87	452;36;1156;947
[18] FSS	G	1172;4;4;507	1502;6;43;120	1004;23;2;283	1052;1;749;1117
[18] FSS	G + M	1125;132;248;1495	258;112;360;870	100;172;1;505	67;1;621;741
[18] FSS	G + M + I	352;61;97;1478	159;319;364;212	165;153;1;1396	1033;3;590;71
[18] FSS	w(G + M + I)	1;1;10;1272	**1;1;3;241**	1;1;1;851	220;2;837;56

limiting the scalability of the approach. Trex [60] and the GNN variant of Li et al. [40] provide the second-best results.

However, the FSS models with custom weights has surprisingly the highest MRR10 for the x64 comparison vs. Netgear R7000. We used the weights shipped in the code, which have been optimized for the OpenSSL comparison. This proves that the optimization process that FSS implements has practical use cases, however it does not extend to other configurations. Table 7 also shows comparisons across different architectures, in particular the ARM32 column for Netgear and the MIPS32 one for TP-Link show the same-architecture comparison. The Netgear R7000 firmware is compiled for ARM 32 bit, while the TP-Link Deco-M4 for MIPS 32 bit: this shows why Asm2Vec has high MRR10 values in the corresponding columns. Finally, Table 6 contains the actual ranking results of the vulnerable functions for the Netgear R7000 image, showing that quite high MRR10 values may hide quite low rankings in practice.

5 Discussion

We now draw some conclusions from the previous results and answer a number of research questions.

Results show that one machine-learning model, the GNN from Li et al. [40], outperforms all the other variants in the six evaluated tasks, achieving performances similar to the less-scalable GMN version. Other embeddings-based models [45, 49, 60, 76] show lower but similar accuracy. Zeek [67], which is a direct-comparison approach, has higher AUC on large functions. Asm2Vec [14] does not perform any better than other models, and fuzzy hashing approaches are not effective when multiple compilation variables change at the same time.

Which are the main contributions of the novel machine-learning solutions compared to simpler fuzzy hashing approaches? Deep-learning models provide an effective way of learning a function representation (i.e., an embedding), forcing a spatial separation between different classes of functions. Differently from fuzzy hashing approaches, machine-learning models achieve high accuracy even when multiple compilation variables change at the same time and they benefit from the advantage of large training datasets built on top of a reliable ground truth defined by the compilation options. The usage of a Siamese architecture [40, 45, 49, 76] in combination with a margin based loss [40, 79] introduced significant improvements in the results. Moreover, GNNs [40, 45, 76, 79] are an effective function encoder that can be used in combination with other instruction level encoders [45, 79].

Which is the role of different sets of features? Results show that the choice of the type of machine-learning model, in particular the GNN, and the loss functions are as important as the features in input. Using basic block features (e.g., ACFG) provides better results, but there is a minimal difference between carefully manually engineered features and simpler ones, such as the bag of words of the basic block opcodes. Surprisingly, in-

Table 7: Vulnerability test.

		Netgear R7000				TP-Link Deco-M4			
Model name	Description	x86	x64	ARM32	MIPS32	x86	x64	ARM32	MIPS32
[67] Zeek (direct comparison)	Strands	0.13	0.15	0.27	0.05	0.07	0.24	0.14	0.22
[40] GMN (direct comparison)	CFG + BoW opc 200	**0.88**	0.54	**1.00**	**0.79**	**0.67**	**0.73**	**0.70**	**0.78**
[40] GMN (direct comparison)	CFG + No features	0.65	0.43	0.69	0.54	0.44	0.47	0.32	0.32
[40] GNN	CFG + BoW opc 200	0.33	0.32	0.56	0.30	0.49	0.56	0.36	0.61
[40] GNN	CFG + No features	0.25	0.03	0.53	0.25	0.22	0.20	0.12	0.27
[76] GNN (s2v)	CFG + BoW opc 200	0.33	0.31	0.67	0.34	0.39	0.28	0.36	0.59
[76] GNN (s2v)	CFG + Gemini	0.33	0.04	0.38	0.25	0.11	0.26	0.28	0.11
[76] GNN (s2v)	CFG + No features	0.07	0.00	0.03	0.00	0.00	0.00	0.06	0.00
[45] w2v + AVG + GNN (s2v)	CFG + N. asm 150	0.10	0.08	0.50	0.00	0.05	0.06	0.03	0.18
[45] w2v + wAVG + GNN (s2v)	CFG + N. asm 150	0.05	0.03	0.04	0.00	0.00	0.03	0.04	0.27
[45] w2v + RNN + GNN (s2v)	CFG + N. asm 150	0.00	0.00	0.03	0.00	0.03	0.11	0.08	0.14
[49] w2v + SAFE	N. asm 150	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
[49] w2v + SAFE	N. asm 250	0.00	0.00	0.00	0.00	0.04	0.02	0.07	0.03
[49] w2v + SAFE + trainable	N. asm 150	0.08	0.29	0.29	0.16	0.04	0.16	0.24	0.09
[49] rand + SAFE + trainable	N. asm 150	0.00	0.00	0.00	0.03	0.06	0.16	0.11	0.07
[60] Trex	512 Tokens	0.40	0.48	0.44	0.50	0.29	0.42	0.22	0.61
[14] Asm2Vec	Rand walks asm	0.25	0.13	0.88	0.50	0.11	0.11	0.02	0.67
[38] PV-DM	Rand walks asm	0.31	0.00	0.75	0.25	0.00	0.00	0.00	0.32
[38] PV-DBOW	Rand walks asm	0.07	0.00	0.63	0.28	0.04	0.00	0.00	0.34
[74] Catalog_1	size 16	0.00	0.00	0.06	0.00	0.00	0.00	0.00	0.05
[74] Catalog_1	size 128	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.57
[18] FSS	G	0.13	0.04	0.13	0.25	0.04	0.00	0.07	0.03
[18] FSS	G + M	0.00	0.00	0.25	0.25	0.02	0.06	0.00	0.10
[18] FSS	G + M + I	0.00	0.00	0.25	0.08	0.11	0.06	0.00	0.06
[18] FSS	w(G + M + I)	0.53	**0.58**	0.75	0.13	0.06	0.00	0.00	0.00

Table 8: Models comparison on input features size, number of parameters, batch size, number of training epochs and training time.

	Description	Feature size	NN params	Batch size	Train epochs	Train/epoch (s)
[67] Zeek (direct comparison)	Strands	1024	3,345,032	32	10	160
[40] GMN (direct comparison)	CFG + BoW opc 200	200	181,634	20	16	1,026
[40] GMN (direct comparison)	CFG + No features	7	163,106	20	16	857
[40] GNN	CFG + BoW opc 200	200	172,418	20	10	944
[40] GNN	CFG + No features	7	153,890	20	10	767
[76] GNN (s2v)	CFG + BoW opc 200	200	75,266	250	5	1,238
[76] GNN (s2v)	CFG + Gemini	7	38,210	250	5	1,068
[76] GNN (s2v)	CFG + No features	7	38,210	250	5	555
[45] w2v + AVG + GNN (s2v)	CFG + N. asm 150	100	28,002,066	250	5	2,254
[45] w2v + wAVG + GNN (s2v)	CFG + N. asm 150	100	28,002,966	250	5	2,140
[45] w2v + RNN + GNN (s2v)	CFG + N. asm 150	100	28,966,866	250	7	8,628
[49] w2v + SAFE	N. asm 150	100	41,377,002	250	5	491
[49] w2v + SAFE	N. asm 250	100	41,377,002	250	5	789
[49] w2v + SAFE + trainable	N. asm 150	100	97,269,002	250	10	553
[49] rand + SAFE + trainable	N. asm 150	100	97,269,002	250	10	533
[14] Asm2Vec	Rand walks asm	200	-	-	10	1,096
[38] PV-DM	Rand walks asm	200	-	-	10	966
[38] PV-DBOW	Rand walks asm	200	-	-	10	549

struction embeddings [45] do not boost the performances of the GNN models, however we think that extensive testing is needed to evaluate other possible combinations. Zeek [67] shows how dataflow information can boost the results, especially for large functions. Finally, fuzzy hashing approaches are more sensitive to the type of features, due to the lack of a training phase.

Do different approaches work better at different tasks? In particular, is the cross-architecture comparison more difficult than working with a single architecture? Our evaluation shows that most of the machine-learning models perform very similarly on all the evaluated tasks, both in the same and cross architectures. Moreover, it is not necessary to train them on a specific task, since using the most generic task data (XM) allows to achieve performances that are overall close to the best for each task. This is not the case for fuzzy hashing methods. For instance, FunctionSimSearch graphlet features have sim-

ilar performances in all the tasks, but their combination with others does decrease the AUC in some tasks. However, not all the approaches can be used in a cross-architectures comparison: Asm2Vec [14] and the two paragraph2vec [38] models are limited to the same-architecture comparisons, due to the specific unsupervised training approach, as well as Catalog1 [74].

Is there any specific line of research that looks more promising as a future direction for designing new techniques? Results show that deep-learning models have the scalability and precision requirements for the different function similarity tasks, especially due to the ability to learn a function representation suitable to multiple tasks. Although the GNN models provided the best results, there are tens of different variants that need to be tested. Moreover, the combination of GNN with assembly instruction encoders is another promising direction [45, 78, 79]. Many of the previous works have focused their effort on selecting different features and feature abstraction levels, but most recent machine-learning models only use the normalized assembly code or an intermediate representation, leveraging the power of representation learning. The effects of combining intermediate representations and dataflow information must be studied as well. Furthermore, we have observed that the selection of features and machine-learning models are not the only aspects that influence the performance of an approach. Some of these complementary aspects such as the training strategy and loss functions have been barely discussed in the past and only recently explored. Li et al. [40] introduced two alternative loss functions, an Euclidean margin-based one, and an approximated Hamming similarity for efficient nearest neighbor search. Following a similar direction, the latest research of CodeCMR [79] shows a significant improvement in the results as a consequence of adopting a norm weighted sampling method (a type of distance weighted sampling [73]) in combination with a triplet loss [66].

6 Conclusions

This paper performs the first measurement study covering more than five years of research works tackling binary function similarity. We identified a number of challenges in the research field, and how they make meaningful comparison difficult, if not outright impossible. Our work aims at bridging this gap and helping the community gain clarity in this research field. We hope that by releasing all our implementations, datasets, and raw results, the community will have a reference point to start building new approaches and will be encouraged to evaluate them against a common framework to better discern which novel aspects are actually improving the state of the art, and which aspects just *appear* to do so.

7 Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions. The work was supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 771844 BitCrumbs).

References

[1] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71, 2015.

[2] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Fossil: a resilient and efficient system for identifying foss functions in malware binaries. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):1–34, 2018.

[3] ANSSI. Machoc hash. https://github.com/ANSSI-FR/polichombr/blob/dev/docs/MACHOC_HASH.md.

[4] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

[5] Jane Bromley, James W Bentz, Léon Bottou, Isabelle Guyon, Yann LeCun, Cliff Moore, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(04):669–688, 1993.

[6] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. BinGo: cross-architecture cross-OS binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 678–689, Seattle, WA, USA, 2016. ACM Press.

[7] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[8] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711, 2016.

[9] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 266–280, New York, NY, USA, 2016. Association for Computing Machinery.

[10] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*, pages 79–94, Barcelona, Spain, 2017. ACM Press.

[11] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*, pages 349–360, Edinburgh, United Kingdom, 2013. ACM Press.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[13] Diaphora. Joxean Koret. https://github.com/joxeankoret/diaphora.

[14] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489, San Francisco, CA, USA, May 2019. IEEE.

[15] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 461–470, 2016.

[16] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In *Proceedings of Network and Distributed System Security Symposium*, 2020.

[17] Thomas Dullien. FunctionSimSearch. https://github.com/googleprojectzero/functionsimsearch.

[18] Thomas Dullien. Searching statically-linked vulnerable library functions in executable code . https://googleprojectzero.blogspot.com/2018/12/searching-statically-linked-vulnerable.html.

[19] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, San Diego, CA, August 2014. USENIX Association.

[20] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA, 2016. Internet Society.

[21] Li et al. graph_matching_networks.ipynb. https://github.com/deepmind/deepmind-research/tree/master/graph_matching_networks.

[22] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *International Conference on Software Security and Reliability (SERE)*, 2014.

[23] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 346–359, 2017.

[24] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, Vienna Austria, October 2016. ACM.

[25] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, pages 896–899, Montpellier, France, 2018. ACM Press.

[26] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[27] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Comput. Surv.*, 54(3), apr 2021.

[28] Hex-Rays. Ida pro. https://hex-rays.com/ida-pro/.

[29] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Cross-architecture binary semantics understanding via similar code comparison. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 57–67. IEEE, 2016.

[30] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Binary Code Clone Detection across Architectures and Compiling Configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 88–98, Buenos Aires, Argentina, May 2017. IEEE.

[31] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. Binmatch: A semantics-based hybrid approach on binary code clone analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 104–114. IEEE, 2018.

[32] He Huang, Amr M Youssef, and Mourad Debbabi. Binsequence: fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 155–166, 2017.

[33] Intel. x86 Instruction Set Reference - MOV. https://c9x.me/x86/html/file_module_x86_id_176.html.

[34] Ulf Kargén and Nahid Shahmehri. Towards robust instruction-level trace alignment of binary code. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[35] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 329–338. IEEE, 2013.

[36] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel San, and Yongdae Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *arXiv preprint arXiv:2011.10749v1*, 2020.

[37] Nathaniel Lageman, Eric D Kilmer, Robert J Walls, and Patrick D McDaniel. Bindnn: Resilient function matching using deep learning. In *International Conference on Security and Privacy in Communication Systems*, pages 517–537. Springer, 2016.

[38] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[39] Yuancheng Li, Boyan Wang, and Baiji Hu. Semantically find similar binary codes with mixed key instruction sequence. *Information and Software Technology*, page 106320, 2020.

[40] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*, pages 3835–3845. PMLR, 2019.

[41] Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. Gated graph sequence neural networks. In *Proceedings of ICLR'16*, April 2016.

[42] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. 2017.

[43] Xiang Ling, Lingfei Wu, Saizhuo Wang, Tengfei Ma, Fangli Xu, Alex X Liu, Chunming Wu, and Shouling Ji. Hierarchical graph matching networks for deep graph similarity learning. *arXiv preprint arXiv:2007.04395*, 2020.

[44] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 667–678, 2018.

[45] Luca Massarelli, Giuseppe A. Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. In *Proceedings 2019 Workshop on Binary Analysis Research*, San Diego, CA, 2019. Internet Society.

[46] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400, 2014.

[47] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How Machine Learning Is Solving the Binary Function Similarity Problem — Artifacts and Additional Technical Details. https://github.com/Cisco-Talos/binary_function_similarity.

[48] Luca Massarelli. Code for the paper Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. https://github.com/lucamassarelli/Unsupervised-Features-Learning-For-Binary-Similarity.

[49] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Safe: Self-attentive function embeddings for binary similarity. In *Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2019.

[50] Luca Massarelli and Giuseppe Antonio Di Luna. Code for the paper SAFE: Self-Attentive Function Embeddings for binary similarity. https://github.com/gadiluna/SAFE.

[51] Elie Mengin. Weisfeiler-Lehman Graph Kernel for Binary Function Analysis. https://blog.quarkslab.com/weisfeiler-lehman-graph-kernel-for-binary-function-analysis.html.

[52] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. 2013.

[53] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26:3111–3119, 2013.

[54] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 253–270, Vancouver, BC, August 2017. USENIX Association.

[55] Jiang Ming, Dongpeng Xu, and Dinghao Wu. Memoized semantics-based binary diffing with application to malware lineage inference. In *IFIP International Information Security and Privacy Conference*, pages 416–430. Springer, 2015.

[56] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. Binsign: fingerprinting binary functions to support automated analysis of code executables. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2017.

[57] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[58] Fabio Pagani, Matteo Dell'Amico, and Davide Balzarotti. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 354–365, 2018.

[59] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Code for the paper TREX: Learning Execution Semantics from Micro-Traces for Binary Similarity. https://github.com/CUMLSec/trex.

[60] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[61] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *IEEE Symposium on Security and Privacy*, 2015.

[62] Jing Qiu, Xiaohong Su, and Peijun Ma. Library functions identification in binary code by using graph isomorphism testings. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 261–270. IEEE, 2015.

[63] Nguyen Anh Quynh. Capstone - The Ultimate Disassembler. https://www.capstone-engine.org/.

[64] Kimberly Redmond, Lannan Luo, and Qiang Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. In *NDSS Workshop on Binary Analysis Research (BAR)*, 2019.

[65] Radim Rehurek and Petr Sojka. Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, 3(2), 2011.

[66] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[67] Noam Shalev and Nimrod Partush. Binary Similarity Detection Using Machine Learning. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security - PLAS '18*, pages 42–47, Toronto, Canada, 2018. ACM Press.

[68] Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape: Scalable and robust binary library function identification using function shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 301–324. Springer, 2017.

[69] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[70] Tencent. BinaryAI Python SDK. https://github.com/binaryai/sdk.

[71] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[72] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 319–330. IEEE, 2017.

[73] Chao-Yuan Wu, R Manmatha, Alexander J Smola, and Philipp Krahenbuhl. Sampling matters in deep embedding learning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2840–2848, 2017.

[74] xorpd. FCatalog. https://www.xorpd.net/pages/fcatalog.html.

[75] Xiaojun Xu. DNN Binary Code Similarity Detection. https://github.com/xiaojunxu/dnn-binary-code-similarity.

[76] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 363–376, New York, NY, USA, 2017. Association for Computing Machinery.

[77] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Transactions on Software Engineering*, 45(11):1125–1149, 2018.

[78] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou ouang, and Shi Wu. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(01):1145–1152, April 2020.

[79] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems*, 33, 2020.

[80] Xiaochuan Zhang, Wenjie Sun, Jianmin Pang, Fudong Liu, and Zhen Ma. Similarity Metric Method for Binary Basic Blocks of Cross-Instruction Set Architecture. In *Proceedings 2020 Workshop on Binary Analysis Research*, San Diego, CA, 2020. Internet Society.

[81] Dongdong Zhao, Hong Lin, Linjun Ran, Mushuai Han, Jing Tian, Liping Lu, Shengwu Xiong, and Jianwen Xiang. Cvsksa: cross-architecture vulnerability search in firmware based on knn-svm and attributed control flow graph. *Software Quality Journal*, 27(3):1045–1068, 2019.

[82] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019. arXiv: 1808.04706.

[83] Zynamics. BinDiff. https://www.zynamics.com/bindiff.html.