Malware and Vulnerability Analysis using Graph-synchronized Language Model

Paventhan Vivekanandan
Indiana University
Bloomington, USA
pvivekan@iu.edu

Alexander Shroyer
Indiana University
Bloomington, USA
ashroyer@iu.edu

Martin Swany
Indiana University
Bloomington, USA
swany@iu.edu

***Abstract*—Malware and Vulnerability analysis are integral components of Cybersecurity. Malware authors employ code obfuscation techniques such as Control-flow flattening and Virtualization to escape detection from antivirus tools. Also, vulnerable code analysis gets complicated when the code is optimized using compiler flags. This paper proposes a binary function similarity detection (BFSD) framework that combines traditional Graph Neural Networks and relatively more recent Large Language Models using ensemble techniques to break code obfuscation and code optimization problems. The framework projects different facets of a binary program, such as control-flow graphs and assembly codes, into different Neural Network architectures, such as GNN and LLM, synchronizes their training process by maintaining the same testing and training data, and finally combines the predictions using ensemble techniques. The diverse features employed by the machine learning models expose unique subsets of functions, and the ensemble takes advantage of this. Experiments show state-of-the-art accuracy in breaking both code obfuscation and code optimization. We use two transformations for code obfuscation: *Control-flow Flattening* and *Virtualization*. We use three compiler flags for code optimization: O_1, O_2, and O_3. We also evaluate the robustness of the framework by cascading *Mixed-Boolean Arithmetic* with *Flatten*.**

***Index Terms*—Control-flow Flattening, Virtualization, Mixed-Boolean Arithmetic, Compiler Optimization, Binary Function Similarity Detection, Software Reverse Engineering.**

I. Introduction

Malware authors widely use code obfuscation mechanisms to prevent their code from being detected by antivirus software [1] [2] [3] [4]. This paper investigates three popular and hard-to-break code obfuscation mechanisms: *Control-Flow Flattening*, *Virtualization*, and *Mixed-Boolean Arithmetic* (EncodeArithmetic). We attack Flatten and Virtualize directly and then cascade EncodeArithmetic with Flatten for robustness experiments (Sec. V-E). Some of the real-world examples of obfuscation include the *Emotet* banking trojan, which used Control-flow Flattening to evade detection by antivirus tools [5] [6]. Also, .NET malware *DoubleZero* wiper discovered in March 2022 used this transformation [7] and an earlier version of *Kazuar* backdoor employed Control-flow Flattening [7] [8] as well. *Clampi* trojan, seen in 2009, employed Virtualization [9]. More recently, *Wslink*, a malicious loader, also used a Virtualization obfuscator [10]. Flatten obfuscates the code by splitting the basic blocks and placing them side-by-side at the same level using a dispatcher node (e.g., a switch statement), which makes it harder to follow the control-flow pattern [11] [12] [13]. Virtualize turns the code into a specialized bytecode array and replaces the original code with the bytecode and an interpreter [11] [1] [2]. At runtime, the interpreter reproduces the original code's behavior using the bytecode array. Here, each interpreter is unique in terms of its pattern of execution and its code structure. Flatten and Virtualize massively increase the dataset size by introducing superfluous assembly contents and control-flow structures. In our dataset, Flatten increased the count of basic blocks by more than 97%, and Virtualize increased the count by more than 170%.

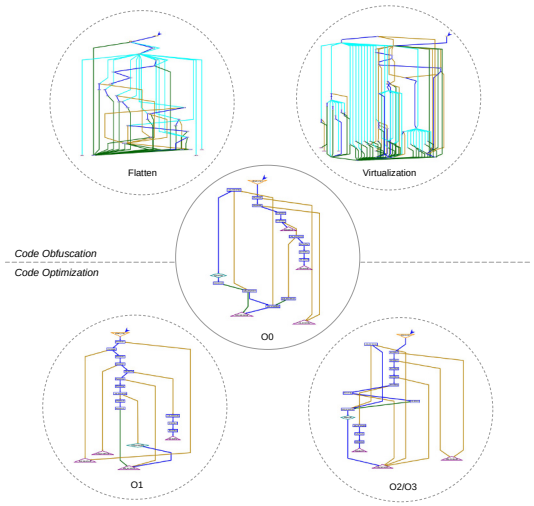


Fig. 1: Transformations for *Mirai* Botnet attack_start()

Another problem is software vulnerability. Software developers reuse existing code components from open-source projects and third-party libraries and incorporate them into their current projects with little or no modification [14] [15]. The reused codes are often compiled using different optimization flags, leading to binary variants of the same function existing across different software systems. Reused code opens the gateway for software vulnerabilities and bug propagation. For example, the *Heartbleed* vulnerability in the OpenSSL library, widely used by applications, provided a backdoor for attackers to steal sensitive data such as passwords, private

keys, and other encrypted data [16]. For code optimization, we analyze three compiler optimization flags: O_1, O_2, and O_3. Unlike code obfuscation, code optimization flags don't usually introduce superfluous contents or dummy assembly words. However, they have a different effect, like reordering the assembly instructions inside a basic block and disturbing the contextual semantics that capture the relationship between adjacent instructions. Code optimization flags aim to improve the performance of the code at the cost of compilation time. The code size may increase depending on the optimization flag used. In our dataset, when applying O_1 and O_2, the basic block count decreased by 0.12% and 2%, respectively. For O_3, the basic block count increased by 20%. The additional contents introduced by O_3 aim at code optimization as opposed to the additional contents introduced by code obfuscation, which aim at concealing information. Fig. 1 shows the obfuscation and optimization variants of attack_start() function of *Mirai* Botnet. For attack_start(), O_3 produces the same code as O_2.

A BFSD tool that can identify malware components from existing attack toolkits, even when obfuscated (or optimized), will greatly help reverse engineers and security administrators. *Apprentice* and *Journeyman* hackers, who likely comprise the largest number of attackers, reuse attack toolkits with some (obfuscation) or no modifications [17]. For example, *Duqu* rootkit [18] has similarity to *Stuxnet* worm and is believed to have been developed by the same authors. Similarly, *Citadel* trojan has similarity to *Zues* worm code [19]. A binary program has different facets of representation, and a single machine-learning architecture is not capable of capturing information essential for classification expressed by all such facets. Also, the varying facets of the program are complementary, and some features become visible in one representation that are not simply visible in the others. In this paper, we aim to break code obfuscation and code optimization using the combined efforts of Large Language Models (LLM) and Graph Neural Networks (GNN). The recent success of LLMs [20] [21] [22] [23] for BFSD makes them indispensable. Also, the nature of the transformations we target in this paper, which involves heavy changes to Control-flow Graph (CFG) structures, coupled with the long-established role of GNNs [24] [25] [26] [27] [28] [29] [30] for BFSD, makes them ideal candidates to study these transformations. Binary codes have a rich corpus of assembly words and the opportunity to visualize them as graphs (unlike the natural language) in the form of CFGs. This allows us to synchronize the training of both the LLM and the GNN by maintaining the same training and testing dataset and finally combine the predictions using ensemble techniques. We call our approach *BinSimStack* and the underlying machine learning architecture as *Graph-synchronized Language Model* (GsLM).

II. Background and Motivation

BFSD deals with identifying and matching binary variants of a function arising from compilation (of the same source code) using different code optimization flags, different compilers, different instruction set architectures, and compilation

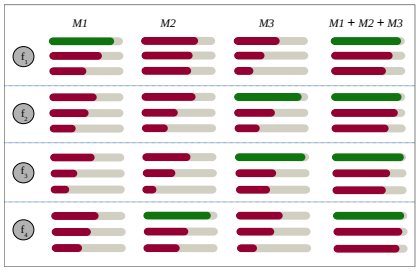


Fig. 2: Subset Preservation

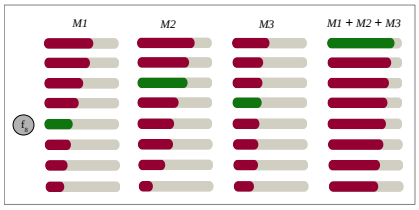


Fig. 3: Steady Gains

from source code transformed using code obfuscation tools. Binary functions provide a rich set of features with varying scopes, which makes them suitable for machine learning. This diverse feature set includes raw binary data, assembly words, tracelets, control-flow graphs, block-flow graphs, instruction counts such as the number of arithmetic instructions, number of logical instructions, number of transfer instructions, etc., unique libC calls, unique callees, unique callers, and many more. Existing papers in the literature extract one or a subset of these features from binaries and attempt to solve one or more transformations such as code optimization, code obfuscation, cross-compiler, and cross-architecture compilation.

When learning in a discriminative setting, a machine learning model only preserves the necessary aspects of input data that will allow it to achieve the highest classification accuracy. This means some aspects of the input data are lost during the training process, and the lost information can still hold valuable details regarding the target classification problem. However, models with different architectures (such as GNN or Transformer) operating on varying input features (based on CFG or assembly words), when trained for the same (BFSD) task, can capture different aspects of the input data. Also, the information lost during the training process of one model is compensated (to some extent) by another model. As a result, when we merge their predictions using an ensemble framework, we may end up with a combined bigger set of correct classifications with better prediction accuracy than the individual models. Here, a *rewarding* merge relies on two key factors: *subset preservation* and *steady gains*.

To understand subset preservation, let us assume the example scenario in Fig. 2, which shows the result of top-3 retrieval given by three models M1, M2, and M3 for four (transformed) functions f_1, f_2, f_3, and f_4 when comparing them against the plain (untransformed) functions in a sample dataset (of

size 8). Here, the green bar indicates the similarity score for the positive (correct) sample, and the red bar indicates the similarity score for the negative (incorrect) samples. Model M1 predicts correctly for the subset $\{f_1\}$ in top-1 while performing poorly for $\{f_2, f_3, f_4\}$, where the correct sample is not retrieved in top-3. Model M2 predicts correctly for the subset $\{f_4\}$ and model M3 predicts correctly for the subset $\{f_2, f_3\}$. Finally, the merge M1 + M2 + M3 preserves the predictions of the individual models and ends up in a bigger set $\{f_1, f_2, f_3, f_4\}$ that correctly classifies all four functions. Here, assume the merge is implemented by simply adding the similarity scores estimated by the individual models for a given function against the samples in the dataset. More merging techniques are discussed in Sec. IV-C step 4. If the individual models predict with high similarity scores on the samples they are predicting correctly, they can compensate for the bad performance of other models, and the final similarity score of the correct samples will be retained in top-1 after the merge. For steady gains, a function keeps gaining its similarity score steadily across predictions of different models and ends up being the top prediction after the merge. Fig. 3 gives the similarity score for top-8 retrievals by M1, M2, and M3 when comparing a (transformed) function f_8 against the (plain) functions in the dataset. The green bar indicates the score of the correct sample, and the red bar indicates the score of the incorrect sample. Here, f_8 is not retrieved in top-1 by any of the individual models. But, f_8 still has decent scores (within top-5) from the individual models, and when merging, the scores add up and push f_8 to top-1 in M1 + M2 + M3.

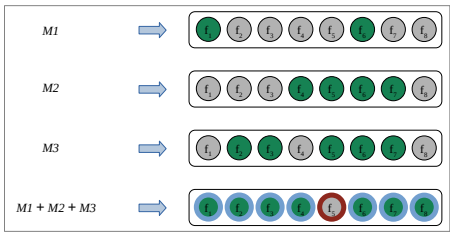


Fig. 4: Combining Predictions

Let's summarize subset preservation and steady gains and introduce some useful terms. Fig. 4 gives the final predictions of all the models. The nodes f_i denote functions in the dataset (total 8). The green nodes indicate the functions successfully retrieved in top-1 when comparing a transformed version of it against the plain (untransformed) version of the functions in the dataset. The grey nodes denote unsuccessful retrieval in top-1. When we combine the predictions, we observe *concords* and *conflicts* during the merge. Here, we define *concord* as the case where a function retrieved successfully in top-1 as part of at least one constituent model is retrieved or preserved successfully in top-1 after the merge. We define *conflict* as the case where a function retrieved successfully in top-1 as part of at least one constituent model is not retrieved or preserved successfully in top-1 after the merge. For example, in Fig. 4, the cases $f_1, f_2, f_3, f_4, f_6, f_7$ (subset preservation) are concords and the case f_5 is a conflict. The node f_8 is a special case (steady gains) where a function not retrieved successfully in top-1 as part of any constituent models is retrieved successfully in top-1 after the merge. Such cases are unique contributions of the merge. Overall, we call the merge *rewarding* if the number of green nodes (implying top-k accuracy) is greater for the merge when compared to all the constituent models.

III. Related Work

Binary functions provide ample and widely varying features that are helpful for similarity detection, and many works in the literature take advantage of this [31]. α*Diff* [32] trains a Convolutional Neural Network on the raw bytes of a function to produce a vector embedding, which it uses to calculate the distance between functions. It also uses function call graph details as features for learning the similarity. *Asm2vec* [33] builds a clone search engine using a *PV-DM* model (a variant of word2vec model [34]) trained on assembly instructions. It uses a sliding window of a specific size and sequentially applies it to the assembly instructions of a given function. Here, the instruction in the middle acts as a target, and the other instructions in the window act as context. While moving the window, it models the target distribution using the context. Based on this, it generates a vector embedding for the assembly instructions. *SAFE* [35] involves two steps. Step 1 generates an embedding for each assembly instruction using the *Skip-Gram* model, another variant of word2vec. Unlike PV-DM, a Skip-Gram models the distribution of context words using the target. For step 2, SAFE feeds the instruction embeddings into a Self-Attentive Bidirection Recurrent Neural Network to generate the final function embeddings. *Palmtree* [21] is another assembly-level tool based on Transformer-Encoder [36] architecture. It learns the embedding of assembly instructions based on three components: a Masked Language Model that predicts missing (or corrupted) instructions, a Context Window Prediction (like Asm2vec) which predicts if two given instructions occur within the same window or not, Def-Use Prediction which predicts if instructions were swapped or not based on data flow information. Palmtree uses BERT [37], a pre-trained language model used in NLP tasks, for instruction embedding. *jTrans* [20], also based on BERT, explicitly models the relationship between a jump instruction and its corresponding target address. It uses two levels of embedding: one for instruction tokens and another for the sequential position of the tokens. BERT treats words that are farther in distance as loosely connected. However, in assembly code, jump instructions bind code that is farther in distance in terms of the address used. jTrans attempts to resolve this by allowing parameter sharing between the token and positional embedding for jump instructions.

Genius [26] is another tool for BFSD based on Graphs. It generates an Attributed Control-flow Graph (ACFG) from the Control-flow Graph, where each block in the CFG encodes an array consisting of instruction (opcode) counts. It then converts the ACFG into a numeric feature vector to calculate the

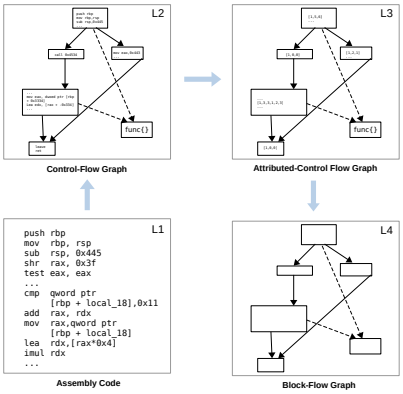


Fig. 5: Layered Representation of a Binary Function

similarity. *Gemini* [24] extends Genius where it uses a Siamese Neural Network based on *Structure2vec* [38] to encode the ACFG into a numeric feature vector. Every architecture uses its own instruction set, which makes the assembly words across them very different. However, the patterns concerning the frequency of instructions used, such as the number of jumps or moves, tend to be invariant or easy to model, and tools such as Genius and Gemini take advantage of this. *HermesSim* [39], the latest addition to BFSD literature, is another graph-based model which uses the P-Code IR of Ghidra. HermesSim captures the necessary execution order between instructions in the assembly code in the form of a novel Semantics-Oriented Graph, which is used for training based on GNN. *BinFinder* [40] attempts to solve code obfuscation and code optimization using features such as VEX-IR (of Angr) encoding, unique libC calls, constants, unique callees, and count of callers and callees. It uses one-hot encoding to capture the VEX-IR instructions, unique libC calls, and constants and tries to see if they are invariant or easily relatable across transformations.

IV. BinSimStack Ensemble Framework

The diversity in the prediction behavior of models arises from various factors, including the diversity in the input feature representation, the difference in the underlying machine learning architecture, and the loss function used. This paper merges four diverse input representations using two varying machine learning architectures: LLM (BERT) and GNN. We use the same loss function, triplet loss, for all the constituent models in the ensemble.

A. Layered Representation of Binaries

For this paper, we choose four layers of representation L_1, L_2, L_3, and L_4 for a binary function (Fig. 5). There is no dependency between the layers. Given the recent success of LLM, the first two layers have been chosen as targets for BERT-based models. For L_1, we use the assembly code. jTrans is a state-of-the-art BERT-based model that produces function-level embedding and allows us to train directly on the assembly code. This makes it a natural choice to operate on L_1. For L_2, we use the CFG. A CFG has assembly words inside basic blocks and explicitly shows the function's control-flow structure. Training on CFG allows us to learn from the assembly and the graph structures simultaneously. Palmtree, another BERT-based model, produces basic-block embeddings that can be encoded inside the CFG and then further trained using a GNN, making it suitable for L_2. For L_3, we use the ACFG of Gemini, which is encoded using the count of string constants, count of numeric constants, count of transfer instructions, count of calls, count of instructions, count of arithmetic instructions, and count of offspring for each basic block. Gemini's intuition for selecting the instruction counts as the feature set is that the count of instructions and offspring tend to be invariant or easily relatable to allow for cross-architecture predictions. We extend this approach to analyze whether the same feature set allows for prediction across code obfuscation and optimization. We use a variant of Gemini (Gemini-trp) to operate on ACFG. For L_4, we drop Gemini's basic-block level features and keep only the count of offspring and the count of calls, which we can generate from the structure of the CFG and the function call graph. Additionally, we use the in-degree and out-degree of nodes to encode the fourth layer. The transformations we target in this paper involve heavy modifications to graph structures. By allowing only structure-based attributes at L_4, we aim to model those transformations without interference from other (non-structure-based) attributes. We build our model named *Cirrina* (Sec. IV-B), which we train on L_4 representation.

B. Cirrina

Structure2vec [38], which is based on probabilistic graphical models, produces embeddings for structured data such as graphs. It uses the graph topology and iteratively learns the representation of every node in the graph by interacting with its neighbors. It learns the embeddings with the help of a Neural Network using shared parameters W_1, W_2 as follows.

$$\mu_i^{(t)} = \sigma(W_1 x_i + W_2 \sum_{j \in \mathcal{N}(i)} \mu_j^{(t-1)}) \quad (1)$$

Here, μ_i denotes the embedding of node i, x_i denotes the label information of i, t represents the iteration number, and $\mathcal{N}(i)$ denotes the neighbors of i. Structure2vec implements $\sigma(.)$ using ReLU. The shape and the training process of equation (1) are inspired by Mean-Field Variational Inference. Structure2vec originally addresses the classification of graphs in a dataset. Here, the node label x_i is initialized using 1-of-K encoding based on the available nodes in the dataset. At iteration zero ($t = 0$), the embedding $\mu_i^{(0)}$ is set to 0 for every node $i \in \mathcal{V}$ (for node set $\mathcal{V}$) in the graph. After the first iteration ($t = 1$), the node-level features of immediate neighbors (1-hop) are aggregated into the embedding $\mu_i^{(1)}$ for node i. After T iterations ($t = T$), the interactions from T-hop neighbors are available and aggregated into the embedding $\mu_i^{(t)}$ for node i. If we chose to terminate after T iterations, then $\{\mu_i^{(t)}\}_{i \in \mathcal{V}}$ are given as the final embeddings for the graph. One observation to consider is that the node level features of

neighbors that are closer (in hop distance) influence the target node more strongly when compared to the farther nodes.

Gemini [24] extends the Structure2vec approach to calculate the similarity between ACFG of binary functions. Gemini makes two important changes to the Structure2vec model. First, it assigns block-level and inter-block attributes to node label x_i. Each node label x_i is designed as a vector based on the count of instructions and offspring (Sec. IV-A). Second, it uses a Siamese architecture to calculate the similarity. It implements the Siamese architecture using two identical Structure2vec networks with the same set of parameters. The two Structure2vec networks take the ACFG of graphs g_1 and g_2 as input respectively and produce embedding $\phi(g_1) = \{\mu_i\}_{i \in \mathcal{V}_1}$ for node set $\mathcal{V}_1$ of g_1 and $\phi(g_2) = \{\mu_i\}_{i \in \mathcal{V}_2}$ for node set $\mathcal{V}_2$ of g_2. It finally uses cosine similarity to calculate the distance between $\phi(g_1)$ and $\phi(g_2)$.

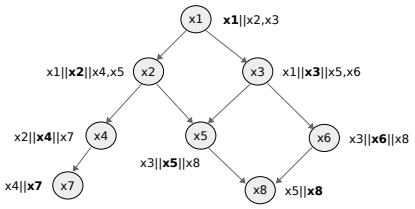


Fig. 6: Cirrina encoding of tracelets as node label

We introduce Cirrina by making three changes to Gemini. First, we use a Siamese Triplet architecture (with three identical Structure2vec networks) and a Triplet Loss function for training instead of the Siamese Twin architecture and the Mean Squared Error (MSE) loss function employed by Gemini. Second, Cirrina works only on structure-based attributes, which are layer L_4 features, and discards the non-structure-based contents of layer L_3. More specifically, the node label x_i for node i now takes attributes such as in-degree, out-degree, count of calls, and count of offspring. For example, in Fig. 6, node $x1$ value is the 4-tuple [0, 2, 0, 7] since in-degree = 0, out-degree = 2, calls = 0, offspring = 7. We can gather all this information from the structure of the CFG and call graph. Third, we extend the node label x_i to include tracelets (partial paths of execution). The tracelets at node i will include the parent of i as the starting node. It then includes i followed by children of i. Here, we use a *busy parent* approach. When a node i has multiple parents, we select the one with maximum in-degree (busy parent), and when there is more than one parent with the same maximum in-degree, we break ties arbitrarily. For example, the tracelet encoding for node $x2$ in Fig. 6 is $x1||x2||x4, x5$. Here, $x1$ is the parent of $x2$, acting as the starting node. Then, we include $x2$ followed by its children $x4, x5$. Also, since $x5$ has the maximum in-degree, it is included as the parent in the encoding for $x8$. We use the tracelet encoding of nodes as the label information during training. In the tracelet encoding, we instantiate each x_i with the corresponding 4-tuple. Cirrina aims to capture the branching structure of a function using tracelets at the node level and then exchanges this information among all the nodes during training. A graph dataset might exhibit learnable patterns at high-level semantics, such as the branching structure of a CFG captured by Cirrina using tracelets, and we aim to exploit it (see appendix). We used tracelets of length $<= 3$, and increasing the length beyond three didn't cause any further improvement to the accuracy of predictions for our dataset. We also limit the number of children in the tracelet encoding of a node to 6.

C. BinSim-Stack Architecture

The BinSim-Stack design (Fig. 7) involves four steps.

Step 1: First, we apply one of the two transformations to the input function: obfuscate the source code using *Tigress* [11] (a source code obfuscator) and compile to produce the corresponding object file or compile the source code directly using one of the three compiler optimization flags O_1, O_2, or O_3. Next, we send the object file as input to a disassembler such as *Angr* to extract the CFG/Assembly. Then, we pre-process the CFG/Assembly to produce four different layers of representations of the input function as given in Sec. IV-A.

Step 2: In step 2, we produce encoding from input CFG/Assembly at each layer of representation. For layer L_1, we use jTrans pre-trained model to produce the input embeddings. For L_2 we use Palmtree. Since Palmtree is a basic block embedding framework, we build a graph encoding from the CFG by replacing its node (assembly) contents using the embedding produced by the original (pre-trained) Palmtree model for respective nodes. For layer L_3, we define and use a new variant of Gemini called *Gemini-trp* (Gemini-Triplet). Gemini-trp involves two changes to Gemini. First, it uses the count of logic instructions instead of string constants. We keep the remaining features the same. We introduced this change to assess the impact of EncodeArithmetic on the ensemble framework (Sec. V-E). Second, Gemini-trp is trained on a Siamese Triplet network using a Triplet Loss function (like Cirrina and Palmtree). Siamese Triplet network showed a significant increase in accuracy over its Siamese Twin counterpart, and this motivated its usage instead of the Siamese Twin network. We use Gemini-trp as the encoding algorithm for layer L_3. For layer L_4, we use Cirrina as the encoding algorithm.

Step 3: In step 3, we use the jTrans Transformer [20] at L_1 and Siamese Triplet with three identical Structure2vec networks at L_2, L_3, and L_4. The anchor to the machine learning model at the respective layers is the Tigress obfuscated (e.g., Flatten or Virtualize) or the compiler optimized (e.g., O_1, O_2, or O_3) version of the original function. We use the plain version of the function (directly compiled without any obfuscation or optimization) as the positive sample and the plain version of another randomly chosen function from the dataset as the negative sample. The positive and negative samples are encoded using the same algorithm used to encode the corresponding anchor. The model outputs two *cosine similarities*: one between the anchor and the positive sample and the other between the anchor and the negative sample. Using cosine similarity (eq. 2), we calculate the *cosine distance*

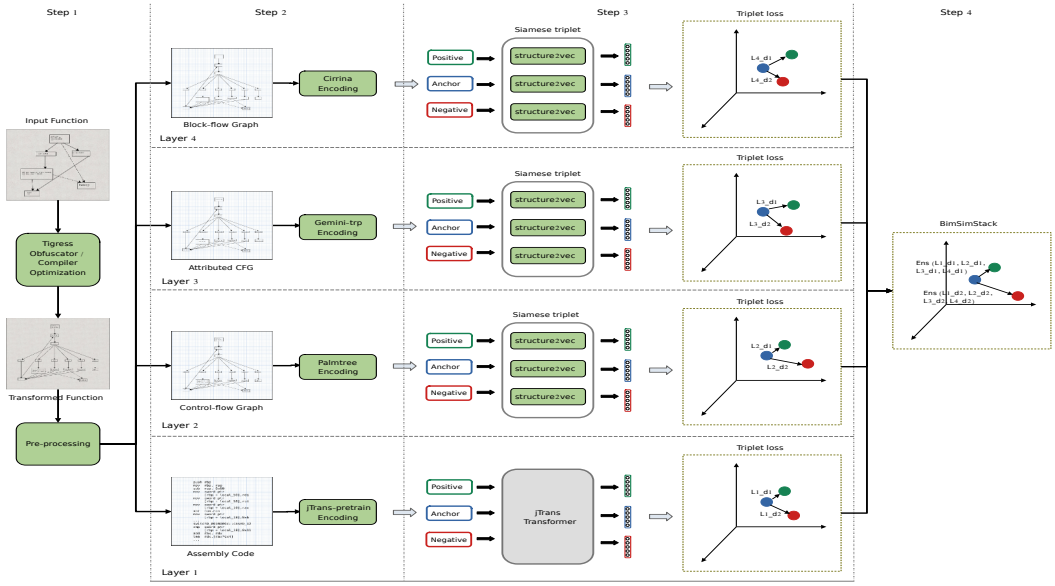


Fig. 7: GsLM Design. The function Ens() is instantiated using one of the ensemble techniques discussed below.

(eq. 3) of the anchor from the positive and the negative samples. We use *Triplet Loss* as the loss function (eq. 4) for the training process and the cosine distance as the distance metric for the loss function.

$$csim(g_x, g_y) = cos(\phi(g_x), \phi(g_y)) = \frac{\langle \phi(g_x), \phi(g_y) \rangle}{\|\phi(g_x)\| . \|\phi(g_y)\|} \quad (2)$$

$$cdis(g_x, g_y) = 1 - csim(g_x, g_y) \quad (3)$$

$$\mathcal{L}_{triplet} = max(cdis(g_a, g_p) - cdis(g_a, g_n) + \alpha, 0) \quad (4)$$

In eq. 2, $\phi(g)$ denotes the output embedding of function g given by the models at respective layers. In eq. 4, α refers to the margin.

Step 4: In step 4, we merge the predictions of models from different layers of representation. The following are the methods used to merge the predictions.

- *Vote* We output the majority vote here. For n models m_1, m_2, ..., m_n, if more than half predict the same outcome, then that will be the final decision.
- *Best Guess* Here, we go with the decision of the model predicting with the highest confidence. For a given triplet $\langle g_a, g_1, g_2 \rangle$, we define the confidence as

$$\mathcal{C} = abs\left(csim(g_a, g_1) - csim(g_a, g_2)\right) \quad (5)$$

- *Aggregate* Here, for triplet $\langle g_a, g_1, g_2 \rangle$, we first add the cosine similarity between g_a and g_1 given by all the models. Then, we add the similarity between g_a and g_2. Finally, we make the decision based on the aggregated similarity measure. Assume n models m_1, m_2, ..., m_n operating at different layers of representation. Then, we define aggregate as follows.

$$csum(g_a, g_x) = \sum_{i=1}^{n} csim(g_a, g_x)_{m_i} \quad (6)$$

$$\mathcal{A}gg. = max(csum(g_a, g_1), csum(g_a, g_2)) \quad (7)$$

For example, if eq. 7 returns $csum(g_a, g_1)$, then the final decision is "g_a is similar to g_1". Also, if x' is the index of the positive sample and $csum(g_a, g_{x'})$ is the highest among all samples in the test dataset, this prediction will be added to recall@1 for aggregate.

- *Weighted Aggregate* This is similar to aggregate. It additionally uses weights for the individual models. We select the weights based on a validation set.

$$w\_csum(g_a, g_x) = \sum_{i=1}^{n} w_i * csim(g_a, g_x)_{m_i} \quad (8)$$

$$\mathcal{W}.agg. = max(w\_csum(g_a, g_1), w\_csum(g_a, g_2)) \quad (9)$$

- *Combined Training* For the final technique, we train all the models together using the same triplet instances.

V. Experiments and Evaluations

Dataset and Baselines: Our dataset includes 17 libraries: *OpenSSL*, *ImageMagick*, *Libxml2*, *Binutils*, *Redis*, *SQLite*, *Curl*, *Musl*, *Libmicrohttpd*, *LibTomCrypt*, *Coreutils*, *Alsa*, *Libmongoc*, *Dbus*, *Allegro*, *Igraph*, *Gsl*. Following [31] [41] [26], we filtered the functions whose CFG had less than five basic blocks. This resulted in 43,048 unique C source code functions, which we used for our analysis. We have given our implementation[1]. We used seven tools as baselines: *Asm2vec* [33], *BinFinder* [40], *Palmtree* [21], *Gemini* [24], *SAFE* [35], *jTrans* [20], and *HermesSim* [39]. In addition to the baseline tools, we also have *Cirrina* and *Gemini-trp*. The choice of baselines is motivated by the diversity of representations used by the tools. Following [40] [20], we used an unofficial

[1]https://github.com/pavenvivek/BinSimStack-RAID-2025

implementation[2] for Asm2vec. For Gemini, we used the author's implementation[3]. For Palmtree, we used the pre-trained model from the author's implementation[4]. We implemented BinFinder from scratch using the original paper. For SAFE, we used the author's implementation[5]. For jTrans, we used the author's pre-trained model to produce the initial embeddings and then used their code[6] for the downstream task of BFSD. For HermesSim, we used the author's implementation[7].

Experimental Setup and Details: For all the experiments (in sec V-A), we used 80% of the dataset for training and 20% for testing. Also, the training and testing sets are kept disjoint for all experiments in the paper. We conducted 33 experiments spanning more than 250 models (including baselines). Sec. V-A contains 25 experiments that analyze five transformations applied to five libraries: *OpenSSL*, *ImageMagick*, *Libxml2*, *Binutils*, and *Redis*. Sec. V-C contains five experiments for scalability and generalizability analysis. Sec. V-D contains one experiment for malware and vulnerability analysis. Sec. V-E contains one experiment that assesses the robustness of the ensemble framework. Sec. V-F contains one experiment that analyzes the relative performance of different ensemble methods.

For code optimization, flag O_1 enables 47 different optimizations [42]. Flag O_2 enables all the O_1 optimizations and 51 optimizations on top of it. Sometimes, O_2 specific optimizations are not compatible with certain functions and, as a result, are not applied to them. For such functions, we end up with the same binary code for O_1 and O_2. For experiments involving O_2, we included only functions whose binary code for O_2 differs from its O_1 counterpart. Similarly, O_3 enables additional optimizations on top of O_2. For experiments involving O_3, we included only functions for which the binary code of O_3 differs from its O_2 counterpart. We used the default parameters of Gemini [24] for Structure2vec network implementation in the Siamese triplet. Palmtree, Gemini-trp, and Cirrina were trained for 100 epochs each. jTrans was trained for 50 epochs. We conducted all the experiments on Ubuntu 22.04 LTS with 64 GB RAM, 32-core Intel i9-13900HX x86_64 CPU architecture, and 8 GB NVIDIA GeForce RTX 4070 GPU. We used Angr as the disassembler for all the tools except HermesSim. We used Ghidra for HermesSim since it depends on Ghidra's P-Code IR. We trained HermesSim for 300 epochs. All the remaining baselines (outside the ensemble) are trained for the default epochs as per their original paper.

Encoding Size vs Training Cost: The encoding size (per basic block) for Palmtree, Gemini-trp, and Cirrina is 200, 7, and 32, respectively. For Palmtree, we faced a trade-off between accuracy and training cost. To determine the encoding size for Palmtree, we compared its performance with other graph-based models with identical architecture, Cirrina and Gemini-trp, and the combined efforts of the three models using the aggregate method (eqn. 7) on a smaller dataset of size 5500. The pre-trained model for Palmtree produced an encoding of size 128 for each assembly instruction. When using the original encoding size of 128 per instruction (for 20 instructions per block), the training time took around 85-90 hours for five transformations, and the storage space required was 33 GB, which is around 600 times of the space needed by Gemini-trp and 200 times of the space needed by Cirrina for the same experiments. The training time for Gemini-trp and Cirrina for the same experiments on dataset size of 5500 is around 1.5 hours each. To reduce the training cost, we decreased the encoding size of Palmtree. We used only the first 10 values of the encoding and included only the first 20 instructions for each basic block. This reduced the encoding size to 200 for each basic block, resulting in an average drop in accuracy by 4% for Palmtree and 3% for the aggregate method for code obfuscation and an average drop of 7.5% for Palmtree and 3% for the aggregate method for code optimization. However, an encoding size of 200 per block reduced the total training time to 6 hours and the storage space to 2 GB for all five transformations combined.

BinSimStack Configuration: For the ensemble approach BinSimStack, we selected jTrans, Palmtree, Gemini-trp, and Cirrina to operate at layers L_1, L_2, L_3, and L_4, respectively. For all the sections below, BinSimStack combines the predictions of individually trained models using the weighted aggregate ensemble method (eqn. 8). The weights are chosen based on validation sets (Table I) and are common to all experiments in the paper.

TABLE I: BinSimStack weights for Weighted Aggregate

	Flatten	**Virtualize**	$\mathbf{O_0 \leftrightarrow O_1}$	$\mathbf{O_0 \leftrightarrow O_2}$	$\mathbf{O_0 \leftrightarrow O_3}$
Cirrina	1	1	0.20	0.20	0.26
Gemini-trp	1	1	0.26	0.26	0.20
Palmtree	1	1	0.15	0.05	0.15
jTrans	1	1	1.15	1.15	1.15

A. Binary Function Similarity Detection

For code obfuscation, we used two Tigress transformations: *Flatten* and *Virtualize* (see appendix for Tigress usage). For code optimization, we used three compiler optimization flags: O_1, O_2, and O_3. For our experiments, we apply one of the above two transformations to an input function. The transformed function as an anchor, along with its original plain version as a positive sample and a different randomly chosen function (plain) as a negative sample, is given as input to the Neural Networks at respective layers during training, and the networks predict the similarity between anchor & positive, and anchor & negative. We use *recall@k* (eqn. 11) predictions to illustrate our evaluations.

Code Obfuscation: For code obfuscation (Table II, III), we see that the BERT-based model jTrans gave the best accuracy among the standalone models, followed by Palmtree. Cirrina performed relatively lower than other models in the

[2]https://github.com/oalieno/asm2vec-pytorch
[3]https://github.com/xiaojunxu/dnn-binary-code-similarity
[4]https://github.com/palmtreemodel/PalmTree
[5]https://github.com/gadiluna/SAFE
[6]https://github.com/vul337/jTrans
[7]https://github.com/NSSL-SJTU/HermesSim

TABLE II: Recall @ top-k analysis for Flatten (avg. testset size = 787)

	OpenSSL			ImageMagick			Libxml2			Binutils			Redis			Average
Models	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**
Asm2vec	11.31	14.78	17.88	30.23	38.53	45.25	27.35	35.42	39.01	11.36	14.35	16.83	35.10	44.26	50.14	23.07
Gemini	29.19	43.33	52.28	28.65	44.26	56.52	13.30	21.22	31.53	30.78	47.00	56.50	29.94	44.55	53.15	26.37
SAFE	0.54	0.63	0.91	1.18	2.37	4.15	0.98	0.98	1.38	0.72	1.65	2.47	1.00	2.14	2.43	0.88
BinFinder	18.43	30.74	38.68	23.12	36.95	46.64	23.16	35.42	44.09	20.45	34.29	44.73	16.18	29.79	37.82	20.26
HermesSim	15.05	24.36	31.75	15.89	22.73	26.76	7.47	14.20	18.98	15.80	24.38	29.64	19.87	31.17	37.86	14.81
Cirrina	45.52	60.94	69.52	30.63	43.67	53.16	37.36	54.85	65.02	31.61	46.28	58.16	33.81	48.28	56.59	35.78
Gemini-trp	71.44	82.75	87.50	66.79	79.64	87.15	59.79	75.18	81.46	69.42	82.33	88.22	64.46	75.35	81.37	66.38
Palmtree	73.35	83.94	87.86	72.92	84.18	89.72	55.75	67.41	73.69	74.58	84.50	89.15	63.32	73.78	80.22	67.98
jTrans	89.59	95.34	97.99	75.88	86.36	92.68	62.18	70.40	72.64	87.80	94.11	96.28	85.24	91.54	93.40	80.13
BinSimStack	**95.80**	**98.35**	**99.08**	**91.50**	**97.43**	**98.41**	**87.29**	**93.12**	**94.61**	**94.42**	**98.96**	**99.48**	**90.97**	**93.98**	**95.98**	**91.99**

TABLE III: Recall @ top-k analysis for Virtualize (avg. testset size = 659)

	OpenSSL			ImageMagick			Libxml2			Binutils			Redis			Average
Models	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**
Asm2vec	0.09	0.28	0.28	1.65	3.08	4.26	1.16	3.03	3.27	0.38	1.41	1.80	2.65	4.15	4.81	1.16
Gemini	4.68	7.77	11.98	4.26	8.53	12.08	5.84	13.78	17.99	6.42	12.21	17.73	8.47	16.44	22.09	5.93
SAFE	0.65	1.21	1.96	0.47	0.94	1.42	0.46	0.70	1.16	0.51	1.02	1.41	0.33	0.66	0.99	0.48
BinFinder	10.29	18.72	25.56	17.77	29.14	42.18	18.92	28.97	35.98	16.19	26.34	34.96	9.96	17.60	26.74	14.62
HermesSim	1.42	2.19	2.76	0.71	1.90	2.62	3.45	5.76	8.06	1.41	2.44	3.09	7.10	10.29	14.70	2.81
Cirrina	6.27	10.76	16.19	10.90	17.53	22.98	7.94	13.78	18.45	6.29	10.79	15.93	7.14	14.61	19.60	7.70
Gemini-trp	10.67	17.79	24.15	11.61	21.80	30.56	14.01	23.81	32.94	9.25	16.06	22.49	12.12	23.58	29.90	11.53
Palmtree	17.69	28.55	37.73	27.48	40.04	51.65	17.28	30.60	39.01	14.39	25.44	32.13	18.77	32.55	39.03	19.12
jTrans	20.59	30.33	38.67	26.54	40.99	49.76	23.13	36.68	43.22	24.03	36.37	44.34	28.90	41.02	48.83	24.63
BinSimStack	**66.47**	**81.08**	**88.48**	**62.08**	**80.33**	**86.01**	**61.21**	**76.63**	**81.54**	**65.68**	**81.61**	**88.17**	**67.77**	**79.06**	**84.38**	**64.64**

TABLE IV: Recall @ top-k analysis for $O_0 \leftrightarrow O_1$ (avg. testset size = 807)

	OpenSSL			ImageMagick			Libxml2			Binutils			Redis			Average
Models	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**
Asm2vec	5.23	8.37	10.56	7.80	11.81	15.40	6.85	10.84	12.83	6.14	9.55	12.51	8.46	12.55	16.22	6.89
Gemini	16.36	25.97	34.15	14.97	26.79	34.59	19.69	30.30	38.05	11.11	17.18	23.25	12.83	19.74	26.23	14.99
SAFE	0.38	0.85	1.23	0.42	0.84	1.26	2.43	4.42	6.85	0.29	0.81	1.18	0.42	1.26	1.55	0.78
BinFinder	18.26	29.11	37.96	17.93	32.48	43.67	20.35	31.63	40.48	12.29	19.03	25.70	15.51	24.11	31.02	16.86
HermesSim	42.15	51.66	56.42	42.61	53.79	59.91	36.72	46.68	54.42	31.18	39.26	43.75	33.54	41.32	44.73	37.24
Cirrina	54.61	64.41	70.40	55.90	64.76	70.67	54.42	70.13	76.54	46.81	56.81	60.81	40.19	47.95	51.62	50.38
Gemini-trp	52.90	66.03	72.78	52.53	65.18	70.88	54.42	63.49	68.80	45.18	55.62	62.74	39.63	48.94	53.59	48.93
Palmtree	33.20	46.71	55.56	37.34	53.37	62.65	27.21	40.04	48.89	20.81	31.48	38.81	14.38	21.86	27.50	26.58
jTrans	80.11	88.96	93.72	72.15	84.59	91.35	69.91	75.22	79.42	79.25	88.00	91.48	76.16	81.66	85.47	75.51
BinSimStack	**87.53**	**94.57**	**96.47**	**84.81**	**93.24**	**95.78**	**77.87**	**84.73**	**88.49**	**85.70**	**91.85**	**94.00**	**80.95**	**85.47**	**87.72**	**83.37**

TABLE V: Recall @ top-k analysis for $O_0 \leftrightarrow O_2$ (avg. testset size = 731)

	OpenSSL			ImageMagick			Libxml2			Binutils			Redis			Average
Models	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**
Asm2vec	4.13	6.76	8.17	7.11	12.00	14.66	5.40	7.29	10.27	4.76	6.77	8.94	8.64	11.72	15.12	6.00
Gemini	11.50	20.78	27.95	17.33	27.77	37.55	13.51	23.78	28.37	6.85	12.70	17.64	14.66	21.60	25.92	12.77
SAFE	0.70	1.00	1.51	0.66	1.33	2.00	2.70	5.13	6.21	0.08	0.41	0.66	0.77	1.08	1.23	0.98
BinFinder	10.89	19.17	26.03	16.44	28.88	36.88	19.72	31.89	42.43	10.86	19.31	26.25	11.26	22.53	28.54	13.83
HermesSim	29.06	37.53	42.28	29.33	37.77	43.11	25.40	35.67	41.08	14.89	22.81	27.35	26.38	35.47	42.60	25.01
Cirrina	35.82	44.39	49.54	49.55	60.44	68.44	44.32	54.59	60.00	28.26	38.79	44.81	36.11	44.59	50.77	38.81
Gemini-trp	38.54	49.24	54.08	50.66	61.77	70.88	34.59	49.72	63.24	29.59	41.47	47.49	36.26	44.29	49.69	37.92
Palmtree	19.97	30.47	37.53	31.77	48.88	58.22	20.27	29.45	35.67	14.21	21.57	25.83	12.96	20.98	26.69	19.83
jTrans	73.15	83.85	87.28	74.88	88.44	93.77	63.24	71.35	74.05	73.91	85.20	88.54	77.00	84.25	86.88	72.43
BinSimStack	**82.34**	**89.70**	**92.02**	**85.77**	**94.44**	**96.88**	**70.54**	**79.18**	**83.51**	**79.26**	**88.12**	**91.80**	**81.01**	**86.57**	**89.19**	**79.78**

ensemble but better than baseline models operating outside the ensemble for Flatten. SAFE performed poorly across all transformations. SAFE performed better on smaller datasets (size < 100), but its accuracy dropped significantly as the size increased. Asm2vec and HermesSim performed poorly on Virtualize, while BinFinder performed better. BinSimStack significantly outperformed all the constituent models and other baseline models across all libraries and all transformations. For example, in Table II, BinSimStack's average recall@1 is 91.99%, around 12 points higher than jTrans's 80.13%, the best among the constituent models. For Virtualize (Table III), BinSimStack's average recall@1 is 64.64%, 40 points higher than jTrans's 24.63%.

Code Optimization: For code optimization (Table IV, V, VI), again, the BERT-based jTrans outperformed other standalone models significantly. Gemini-trp and Cirrina

TABLE VI: Recall @ top-k analysis for $O_0 \leftrightarrow O_3$ (avg. testset size = 274)

	OpenSSL			ImageMagick			Libxml2			Binutils			Redis			Average
Models	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**	**top-2**	**top-3**	**top-1**
Asm2vec	6.06	8.41	9.09	10.88	18.65	23.83	6.09	9.14	9.75	6.61	10.20	12.85	9.47	15.26	18.94	7.82
Gemini	12.12	22.89	31.64	20.72	31.08	40.41	14.63	22.56	31.70	8.88	14.55	19.65	18.94	28.42	33.15	15.05
SAFE	0.67	1.34	1.68	1.55	2.59	5.18	1.84	3.68	4.29	0.75	1.13	1.13	1.05	1.05	2.63	1.17
BinFinder	12.79	19.86	26.26	20.20	31.60	41.96	17.68	29.26	35.36	12.66	21.55	28.54	13.15	22.63	31.57	15.29
HermesSim	26.26	35.01	40.06	33.16	49.22	58.03	29.87	37.80	45.73	17.90	25.14	30.66	20.37	30.86	40.12	25.51
Cirrina	32.99	43.43	51.17	36.26	48.70	61.65	36.58	46.34	56.09	26.46	34.40	40.45	32.10	43.68	47.63	32.87
Gemini-trp	25.58	35.01	41.07	38.86	54.40	63.73	26.82	40.24	44.51	20.03	30.05	37.42	30.00	39.47	46.31	28.25
Palmtree	16.83	24.24	31.64	22.27	35.75	44.04	26.82	36.58	46.95	16.06	23.44	28.35	12.10	20.00	28.42	18.81
jTrans	69.69	82.49	87.87	73.57	85.49	92.22	67.07	81.09	87.80	68.62	80.15	86.76	80.52	91.57	92.10	71.89
BinSimStack	**80.13**	**87.87**	**90.90**	**87.04**	**94.30**	**96.37**	**75.60**	**85.36**	**90.85**	**75.80**	**84.87**	**89.98**	**84.73**	**93.15**	**96.31**	**80.66**

performed similarly and gave the second-best performance. Among the ensemble constituent models, Palmtree gave the lowest performance. Similar to code obfuscation, the ensemble approach BinSimStack significantly outperformed all the constituent models. For example, for $O_0 \leftrightarrow O_1$, BinSimStack average recall@1 is 83.37%, 8 points higher than jTrans (Table IV). For $O_0 \leftrightarrow O_2$, BinSimStack average recall@1 is 79.78%, 7 points higher than jTrans (Table V). For $O_0 \leftrightarrow O_3$, BinSimStack average recall@1 is 80.66%, 9 points higher than jTrans (Table VI). For all 33 experiments (including Sec. V-C, V-D), BinSimStack significantly outperformed all the constituent models; hence, the merge is rewarding for all experiments.

Discussion: Different layers in the ensemble capture different facets of an input binary program, and the contents of those layers are impacted asymmetrically function-wise and to varying degrees by different transformations. For example, Flatten and Virtualize add numerous superfluous control flows and noisy assembly words. However, the superfluous flows and the noisy assembly words have different impacts on different subsets of functions. Cirrina relies more on control flows in terms of in-degree and out-degree than jTrans and Palmtree encoding based on assembly words and Gemini-trp instruction counts, which cover more diverse instructions. Cirrina finds the Tigress transformations more challenging than other models but still contributes uniquely to code obfuscation (Sec. V-B), which means it can predict a subset of functions not covered by jTrans and other models. One reason for this is that on the subset for which Cirrina is more successful, the impact of obfuscation on control flows is relatively lower than on assembly words.

Code optimization, on the other hand, reorders the assembly instructions, disturbing the contextual semantics that capture the relationship between adjacent instructions. The BERT-based jTrans at L_1, which depends heavily on the assembly words, successfully models the reordering effects of the optimization better than any other model. However, Gemini-trp and Cirrina each expose a unique subset of functions not covered by jTrans. Gemini-trp at L_3 uses instruction counts, and any reordering of instructions will not affect Gemini-trp, and Cirrina at L_4 is only dependent on control flows between basic blocks. On the subsets where Gemini-trp and Cirrina were more successful, jTrans finds it more challenging to mitigate the impact of instruction reordering, and the elimination of order relationship between assembly instructions at layers L_3 and L_4 proved to be fruitful. The above discussion shows that the impact of code obfuscation and code optimization on different layers of input representation is asymmetrical function-wise. Thus, jTrans, Palmtree, Gemini-trp, and Cirrina operating on various facets of input representation were relatively more successful on varying subsets of functions across transformations, and by merging those subsets, we end up with a larger set reflecting an improved top-k accuracy.

B. Ablation Study and Unique Contribution Analysis

For the ablation study, we removed the low-performing model from the ensemble and evaluated the performance of BinSimStack. Cirrina and Palmtree were the low-performing (with the least accuracy) ensemble participants for code obfuscation and code optimization, respectively. Out of the 10 experiments for code obfuscation (Table II,III), when Cirrina was removed, BinSimStack dropped in accuracy for all experiments. Out of the 15 experiments for code optimization (Table IV,V,VI), when Palmtree was removed, BinSimStack dropped in accuracy by an average of 1.26 points across 9 experiments and gained in accuracy by an average of 1.94 points across 6 experiments. One important observation is that BinSimStack still significantly outperformed jTrans (the best-performing standalone model) with or without the low-performing model.

The readings of Table VII are derived from the same experiments as in Sec. V-A. In Table VII, the average recall@1 for Cirrina for Flatten is 35.78%, of which 1.29% is unique. This means 1.29% of functions in the test data were correctly predicted only by Cirrina. Except for Virtualize, the LLM model jTrans makes the largest unique contribution across all transformations. The GNN-based models make relatively smaller unique contributions. However, when combined using the ensemble, they add up, significantly boosting the accuracy of the ensemble framework. The unique contributions of GNN-based models show that even though jTrans is trained on the same dataset as the GNN-based models, there are some aspects of program representation that jTrans could not learn through assembly words and parallel graph projections of those same code and training them on GNNs helped the overall learning process remarkably. For Virtualize, the average recall@1 for BinSimStack is 64.64%, of which 25.08%

TABLE VII: Concords, Conflicts, and, Unique Contribution Analysis for recall@1

		OpenSSL		ImageMagick		Libxml2		Binutils		Redis		Average	
Transformation	**Models**	**total**	**unique**	**total**	**unique**	**total**	**unique**	**total**	**unique**	**total**	**unique**	**total**	**unique**
Flatten	Cirrina	45.52	0.18	30.63	0.19	37.36	4.78	31.61	0.61	33.81	0.71	35.78	1.29
	Gemini-trp	71.44	1.00	66.79	3.75	59.79	6.12	69.42	0.61	64.46	1.28	66.38	2.55
	Palmtree	73.35	2.91	72.92	4.34	55.75	2.09	74.58	2.89	63.32	2.00	67.98	2.84
	jTrans	89.59	6.02	75.88	4.54	62.18	5.08	87.80	7.23	85.24	8.02	80.13	6.17
	BinSimStack	95.80	0.82	91.50	3.35	87.29	2.98	94.42	0.72	90.97	1.14	91.99	1.80
	Model Union	95.89	-	92.49	-	87.74	-	95.97	-	91.83	-	92.78	-
	Concords	94.98	-	88.14	-	84.30	-	93.69	-	89.82	-	90.18	-
	Conflicts	0.91	-	4.34	-	3.43	-	2.27	-	2.00	-	2.59	-
Virtualize	Cirrina	6.27	3.18	10.90	4.26	7.94	3.27	6.29	3.08	7.14	3.32	7.70	3.42
	Gemini-trp	10.67	5.52	11.61	4.02	14.01	6.07	9.25	3.85	12.12	5.31	11.53	4.95
	Palmtree	17.69	10.76	27.48	13.74	17.28	9.57	14.39	8.22	18.77	8.97	19.12	10.25
	jTrans	20.59	13.01	26.54	14.45	23.13	13.78	24.03	16.96	28.90	20.09	24.63	15.65
	BinSimStack	66.47	30.52	62.08	18.00	61.21	21.26	65.68	31.23	67.77	24.41	64.64	25.08
	Model Union	43.16	-	54.50	-	46.72	-	42.54	-	50.99	-	47.58	-
	Concords	35.95	-	44.07	-	39.95	-	34.44	-	43.35	-	39.55	-
	Conflicts	7.20	-	10.42	-	6.77	-	8.09	-	7.64	-	8.02	-
$O_0 \leftrightarrow O_1$	Cirrina	54.61	1.14	55.90	2.95	54.42	2.43	46.81	2.74	40.19	0.84	50.38	2.02
	Gemini-trp	52.90	1.71	52.53	2.32	54.42	1.76	45.18	1.48	39.63	0.14	48.93	1.48
	Palmtree	33.20	2.09	37.34	2.74	27.21	2.65	20.81	1.11	14.38	1.12	26.58	1.94
	jTrans	80.11	17.31	72.15	14.55	69.91	12.16	79.25	23.92	76.16	26.93	75.51	18.97
	BinSimStack	87.53	1.90	84.81	2.53	77.87	1.54	85.70	2.88	80.95	3.66	83.37	2.50
	Model Union	89.05	-	90.08	-	82.74	-	87.62	-	79.83	-	85.86	-
	Concords	85.63	-	82.27	-	76.32	-	82.81	-	77.29	-	80.86	-
	Conflicts	3.42	-	7.80	-	6.41	-	4.81	-	2.53	-	4.99	-
$O_0 \leftrightarrow O_2$	Cirrina	35.82	1.81	49.55	2.44	44.32	8.37	28.26	1.75	36.11	1.23	38.81	3.12
	Gemini-trp	38.54	1.61	50.66	2.44	34.59	3.78	29.59	1.58	36.26	0.92	37.92	2.06
	Palmtree	19.97	2.01	31.77	3.55	20.27	1.08	14.21	2.09	12.96	0.92	19.83	1.93
	jTrans	73.15	26.53	74.88	16.88	63.24	20.81	73.91	33.52	77.00	32.40	72.43	26.02
	BinSimStack	82.34	4.84	85.77	2.88	70.54	2.70	79.26	3.59	81.01	2.77	79.78	3.35
	Model Union	81.53	-	89.55	-	82.16	-	81.52	-	82.25	-	83.40	-
	Concords	77.49	-	82.88	-	67.83	-	75.66	-	78.24	-	76.42	-
	Conflicts	4.03	-	6.66	-	14.32	-	5.85	-	4.01	-	6.97	-
$O_0 \leftrightarrow O_3$	Cirrina	32.99	2.35	36.26	3.10	36.58	4.26	26.46	2.45	32.10	1.05	32.87	2.64
	Gemini-trp	25.58	0.33	38.86	4.14	26.82	0.60	20.03	1.70	30.00	1.57	28.25	1.66
	Palmtree	16.83	1.34	22.27	1.03	26.82	4.87	16.06	3.40	12.10	0.52	18.81	2.23
	jTrans	69.69	32.65	73.57	27.97	67.07	24.39	68.62	36.86	80.52	41.57	71.89	32.68
	BinSimStack	80.13	7.07	87.04	3.10	75.60	2.43	75.80	5.67	84.73	4.73	80.66	4.60
	Model Union	78.78	-	87.56	-	82.31	-	78.63	-	85.78	-	82.61	-
	Concords	73.06	-	83.93	-	73.17	-	70.13	-	80.00	-	76.05	-
	Conflicts	5.72	-	3.62	-	9.14	-	8.50	-	5.78	-	6.55	-

is unique. This means when combining the predictions of individual models using BinSimStack, 25.08% of functions incorrectly predicted by individual models now fall into the category of correct predictions. The unique contribution of BinSimStack further increases the accuracy of the ensemble framework. In Table VII, Model Union refers to the percentage of functions in the test data correctly predicted by at least one standalone model in the ensemble framework. Concords refer to the percentage of functions (from Model Union) retained after combining the predictions of standalone models using BinSimStack. Conflicts refer to the percentage of functions missed after combining the predictions of standalone models using BinSimStack. In Table VII, the following relations hold.

BinSimStack = Concords + BinSimStack Unique

Concords = Model Union - Conflicts

The average recall@1 accuracy of BinSimStack for Flatten and Virtualize is 12 and 40 points higher than that of the best-performing standalone model jTrans, respectively. There are two contributing factors to this significant increase in accuracy. One is the unique predictions of the BinSimStack (steady gains), and the other factor is that the BinSimStack can also take advantage of the concords (subset preservation). To understand this better, let's do a case analysis on the prediction accuracy of BinSimStack. Let's assume BinSimStack depends on n models, m_1, m_2, ..., m_n, operating at different layers of representation. Let g_a be the anchor and g_x and g_y be the two samples given as input to the Neural Network at the corresponding layer during testing. The prediction of an individual model m_i, for $i \in \{1, 2, ..., n\}$, depends on eqn. 10.

$$\Delta_{m_i} = csim(g_a, g_x) - csim(g_a, g_y) \tag{10}$$

If we assume x as the index of the positive sample in the test data, then model m_i predicts correctly if $\Delta_{m_i} > 0$. If the same condition is true $\forall y$ in the dataset, where y is the index of a negative sample, then the above prediction will be added to recall@1 for m_i. We define recall as the fraction of similar functions retrieved from the total number of similar functions in the repository (eqn. 11).

$$recall = \frac{\#\ similar\ functions\ retrieved}{\#\ similar\ functions\ in\ repository} \tag{11}$$

Based on the above equations, we have the following possible scenarios. For simplicity, we ignore the case where a model gives no prediction, which happens when both the positive and the negative samples are at the same distance from the anchor.

Case 1: All the n models make the same prediction. So, the models are either predicting correctly or incorrectly. In this case, the BinSimStack will make the same prediction as the participant models. If we assume x as the index of the positive sample, then BinSimStack predicts correctly if $\Delta_{m_{bsim}} > 0$ (where $\Delta_{m_{bsim}}$ defined as in eqn. 12 and $csum$ defined as in eqn. 6), which happens only when $\Delta_{m_i} > 0 \; \forall i \in \{1, 2, ..., n\}$.

$$\begin{aligned} \Delta_{m_{bsim}} &= csum(g_a, g_x) - csum(g_a, g_y) \\ &= \sum_{i=1}^{n} \Delta_{m_i} \end{aligned} \tag{12}$$

Case 2: Of the total n models, j models m_1, m_2, ..., m_j where $j < n$ predict in one direction, and the remaining models m_{j+1},..., m_n predict in the other direction. Also, assume x as the index of the positive sample. If the j models predict correctly, i.e., $\Delta_{m_i} > 0 \; \forall i \in \{1, 2, ..., j\}$, then BinSimStack predicts correctly, i.e., $\Delta_{m_{bsim}} > 0$, only if $\sum_{i=1}^{j} \Delta_{m_i} > 0$ predicts with high confidence such that $\sum_{i=1}^{j} |\Delta_{m_i}| > \sum_{k=j+1}^{n} |\Delta_{m_k}|$.

Example 5.2.1: Let's consider a transformed function f with anchor g_a. Assume we have one positive sample g_x and 100 negative samples in the test dataset. Also, assume that Cirrina predicts correctly for all triplets $\langle g_a, g_x, g_y \rangle$ (where y is the index of negative samples) in the dataset (i.e., the similarity between the anchor and the positive sample is the highest), and Gemini-trp, Palmtree, and jTrans predict incorrectly for some samples. Also, assume that Cirrina predicts with high confidence for all the triplets $\langle g_a, g_x, g_y \rangle$. Then, the prediction for f will be added to recall@1 for both Cirrina and the BinSimStack. As another scenario, consider that Cirrina correctly predicts samples 1 to 20, Gemini-trp correctly predicts samples 20 to 30, Palmtree correctly predicts samples 30 to 50, and jTrans correctly predicts samples 50 to 100. For the given ranges, assume all the models predict with high confidence. Here, f will not be added to recall@1 for the individual models since they are predicting correctly only for the given ranges, but it will be added to recall@1 for BinSimStack (steady gains). This way, BinSimStack gains additionally from models predicting partially but with high confidence.

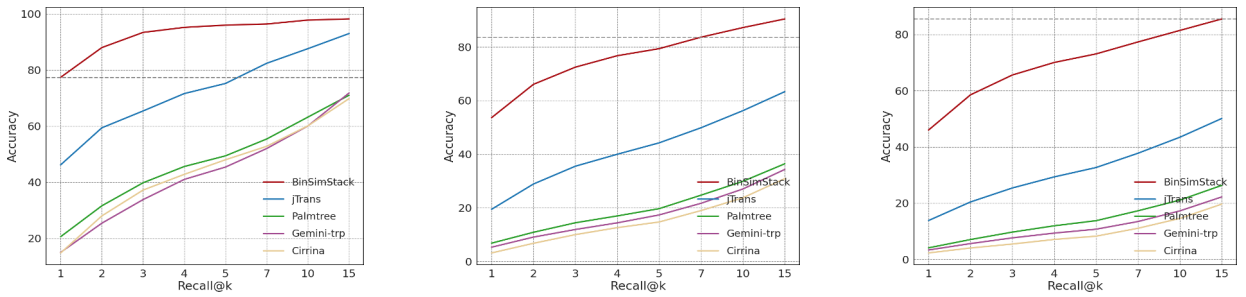
C. Scalability and Generalizability Analysis

For scalability, we evaluate the performance of BinSimStack with respect to the models participating in the framework on three test sets of sizes 500, 3500, and 7500. For generalizability, we evaluate if models trained on one set of libraries can predict functions from new unseen libraries. We build our training set by merging the training data of 5 libraries, OpenSSL, Redis, Binutils, ImageMagick, and Libxml2, from experiments in Sec. V-A. We build a larger testing set from the remaining 12 libraries in our dataset. We don't include functions from the 5 libraries used in the training data for the testing set for generalizability. Using randomly chosen functions, we create evaluation pools of sizes 500, 3500, and 7500 from the testing set. The mid pool (3500) is 7 times bigger than the small pool (500), and the large pool (7500) is 15 times bigger than the small pool. We analyze the performance of BinSimStack and the participant models in the framework by comparing the value of recall@1 for poolsize 500 with recall@7 for poolsize 3500. Since the mid pool is 7 times bigger than the small pool, we assess whether the recall@1 performance of BinSimStack for the small pool matches up with the recall@7 performance of BinSimStack for the mid pool. Similarly, we compare recall@1 for poolsize 500 with recall@15 for poolsize 7500. The dashed line in Fig. 8 represents BinSimStack's prediction for recall@1, recall@7, and recall@15 for poolsize 500, 3500, and 7500, respectively. We use Virtualize for this analysis. Other transformations scale and generalize similarly (see appendix). From Fig. 8, we see that BinSimStack outperforms the participant models significantly. For Virtualize (Fig. 8), recall@1 prediction of BinSimStack for poolsize 500 is 77.4%, recall@7 for poolsize 3500 is 83.62%, and recall@15 for poolsize 7500 is 85.54%, respectively. This shows that the prediction accuracy of BinSimStack is scaling up proportionately with respect to the evaluation poolsize. For poolsize 7500, BinSimStack's recall@10 is 81.46% for Virtualize. This shows that BinSimStack generalizes efficiently when predicting unseen software libraries. The figures also show that the gap between BinSimStack and jTrans keeps increasing with the increase in the evaluation poolsize. In Table VIII, we have given some useful recall@k values and MRR (Mean Reciprocal Rank) values for all transformations on the larger poolsize.

D. Malware and Vulnerability Analysis

In this section, we evaluate the performance of our framework on real-world malware and vulnerability samples. We collected 28 malware samples[8] and 29 vulnerability samples[9][10]. Some malware we added include *Mirai* (Botnet), *Reptile* (Rootkit), *Lyceum* (Backdoor), *Dexter* (Trojan Horse), *Beurk* (Rootkit), and *Gozi* (Trojan Horse). We also included *Heartbleed* vulnerability as part of the samples we tested. We merged the training datasets of the 5 libraries, OpenSSL, Redis, Binutils, ImageMagick, and Libxml2, for Virtualize from experiments in Sec. V-A and trained our model on the larger set. Then, we merged the functions from the testing sets of those 5 libraries with all the functions from the other 12 libraries to create one large testing set. We created an evaluation pool of size 7500 from this large testing set using randomly chosen functions. In Fig. 9, we have given the recall@k analysis for Virtualize, the most vigorous and hardest of transformations to break in our experiments. For this section, each malware and vulnerability sample is individually compared against all the functions in the evaluation pool (of size 7500), and the recall@k statistics are given in Fig. 9a.

[8]https://github.com/vxunderground/MalwareSourceCode
[9]https://www.openssl.org/news/vulnerabilities.html
[10]https://curl.se/docs/security.html

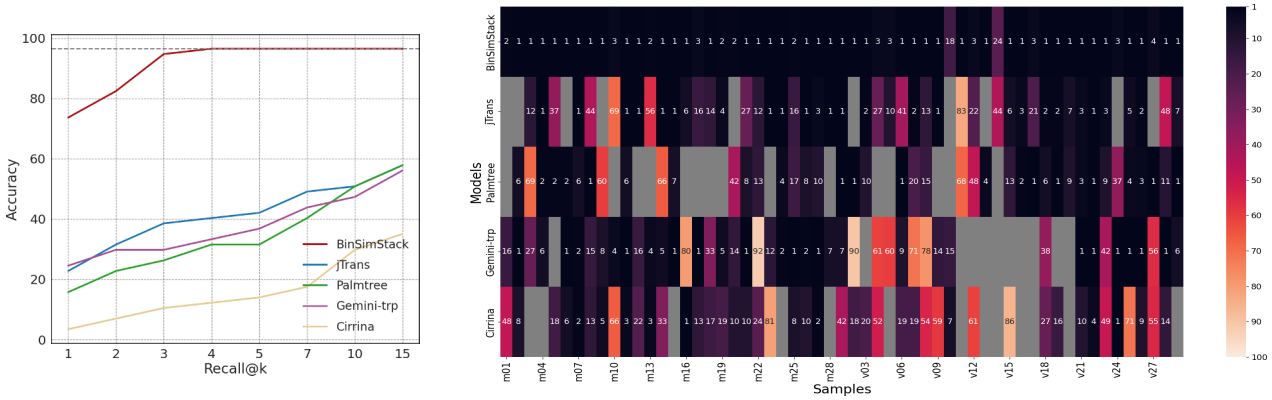


(a) Recall @ top-k (testset size = 500) (b) Recall @ top-k (testset size = 3500) (c) Recall @ top-k (testset size = 7500)

Fig. 8: Scalability and Generalizability analysis for Virtualize

TABLE VIII: Recall@k/MRR for Scalability and Generalizability Analysis

	Flatten			**Virtualize**			**$O_0 \leftrightarrow O_1$**			**$O_0 \leftrightarrow O_2$**			**$O_0 \leftrightarrow O_3$**		
	Poolsize = 7500			**Poolsize = 7500**			**Poolsize = 7500**			**Poolsize = 7500**			**Poolsize = 5000**		
Models	**top-1**	**top-2**	**MRR**	**top-1**	**top-10**	**MRR**	**top-1**	**top-2**	**MRR**	**top-1**	**top-3**	**MRR**	**top-1**	**top-10**	**MRR**
Cirrina	15.69	23.64	26.51	2.28	14.56	6.64	34.57	42.70	43.22	22.49	32.78	29.94	14.79	30.04	20.26
Gemini-trp	46.54	59.59	58.66	3.33	17.25	8.23	33.48	42.30	43.26	20.18	31.62	28.70	11.50	29.78	17.66
Palmtree	52.18	62.57	62.16	4.10	21.09	9.80	8.73	13.44	16.03	4.32	9.89	9.62	1.94	8.54	4.49
jTrans	77.37	85.88	84.12	13.82	43.46	23.43	67.37	77.90	75.86	58.65	75.52	68.64	50.32	79.97	60.79
BinSimStack	**85.60**	**91.66**	**90.11**	**46.02**	**81.46**	**58.21**	**75.81**	**83.57**	**82.07**	**68.70**	**81.80**	**76.41**	**57.69**	**82.00**	**66.58**



(a) Recall @ top-k analysis (b) Sample uniqueness analysis

Fig. 9: Malware and Vulnerability Sample Analysis for Virtualize (testset size = 7500)

Fig. 9b represents the corresponding heatmap. In the heatmap, *m01, m02, ...* refers to the malware samples, and *v01, v02, ...* refers to the vulnerability samples. The values inside the heatmap cells represent the top-k rank, indicating within how many returned predictions the correct input sample falls. The blank grey cells represent cases with a top-k value > 100 indicating bad performance.

In Fig. 9a, the BinSimStack performed significantly better than the standalone models. BinSimStack's recall@1 accuracy is 73.68%, 49 points higher than the best-performing standalone model Gemini-trp. BinSimStack contributed 26.31% uniquely to recall@1. From Fig. 9b, we see that Gemini-trp predicted uniquely at top-1 for the set {*m02, m06, m17, m21, m25, m27, v21, v25, v26, v28*}. Of these, BinSimStack retained {*m02, m06, m21, m25, m27, v21, v25, v26, v28*} (part of concords) in the top-1 while losing *m17, m25* (conflict). The retained set gets added to recall@1 for BinSimStack. Similarly, BinSimStack also gained from the unique contributions of jTrans, Palmtree, and Cirrina. From Fig. 9b, we also see that BinSimStack predicted uniquely at top-1 for the set {*m03, m05, m09, m18, m22, v03, v07, v08, v11, v13, v15, v16, v18, v20, v23*}. Another observation from Fig. 9b is that we see the grey cells (with top-k value > 100), indicating the bad performance of one model, being compensated by other models in the ensemble. When the predictions are merged, the BinSimStack gains significantly, resulting in zero grey cells. This reemphasizes that the models operating on varying representations cover different subsets of functions, and a *rewarding* merge using an ensemble framework can significantly boost the overall prediction accuracy.

E. Robustness of Ensemble Framework

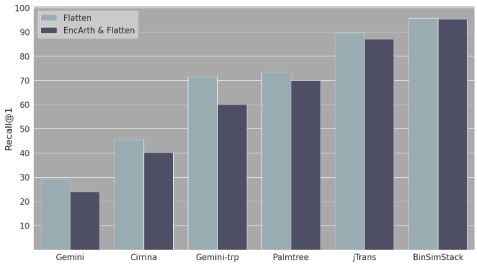


Fig. 10: Robustness Analysis

To assess the robustness of the ensemble, we cascaded the Tigress obfuscation EncodeArithmetic with Flatten and compared it with applying Flatten alone (Table II). We used the OpenSSL library for this experiment. For relative comparison, we kept the training and the testing set the same for the experiment in Table II (Flatten on OpenSSL) and the experiment in this section. EncodeArithmetic obfuscates arithmetic expressions using complex logical operators. Fig. 10 shows that all the ensemble constituent models were negatively impacted. Gemini-trp depends on logical and arithmetic instruction counts and, as a result, is the heavily impacted model. The recall@1 accuracy of Gemini-trp dropped by around 11 points when cascading EncodeArithmetic to Flatten, while the impact on other constituent models was relatively low. Also, the impact on BinSimStack was the lowest. This shows that other layers in the ensemble act as shock absorbers and compensate for the bad performance of one model. Our framework is also effective against cascading more Tigress obfuscations such as RandomizeArgs, EncodeData, EncodeLiterals.

F. Comparing the Ensemble Methods

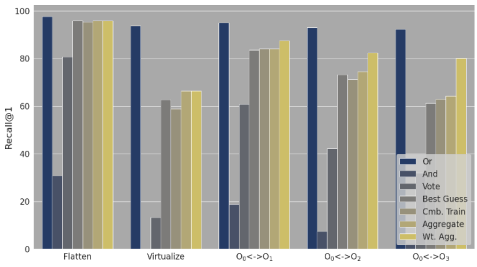


Fig. 11: Recall@1 for ensemble methods

In this section, we compare the performance of the ensemble methods we used in our experiments. For the experiments in Sec. V-A, we have shown the results of the Weighted Aggregate (eqn. 8). Weighted Aggregate gave the best performance overall, followed by Aggregate. This behavior is due to Weighted Aggregate assigning higher weights to the best-performing constituent model jTrans. Best Guess is the third best-performing ensemble method, and Combined Training ended up as the fourth best-performing method, followed by Vote. In Fig. 11, we have given the recall@1 prediction accuracy for different ensemble methods for OpenSSL. We have also provided the predictions for the ensemble method *Or* and *And*. *Or* refers to at least one standalone model (among Cirrina, Gemini-trp, Palmtree, and jTrans) in the ensemble framework predicting correctly (eqn. 10). We cannot use method *Or* because we don't have enough information to know which model makes the correct prediction at runtime. Method *And* refers to the cases where all the models make the same and correct prediction. We can see from the figure that this percentage is the lowest. All the other ensemble methods fall between the prediction accuracy of *And* and *Or*. This shows that the challenge lies in identifying a consistent and effective strategy to combine the predictions of different standalone models at runtime.

VI. Limitations and Future Work

The ensemble depends on standalone models such as Cirrina, Gemini-trp, Palmtree, and jTrans for decision-making. Therefore, any limitations of the standalone models will also impact the ensemble. One such limitation is that jTrans and Palmtree depend on assembly words, which makes them unsuitable for cross-architecture predictions. Moreover, the reverse engineering efforts in this paper start from the successful reconstruction of Assembly/CFG from binaries. So, we also have to factor in the disassembler issues of *Angr* (the disassembler used in this paper) as part of the limitation of the overall process. Also, our tool may not be effective against enryption-based techniques or *Packers*, and more research is needed here.

The cost of training multiple models in the ensemble could be quite expensive. In our work, we observed a high training cost for the layers involving language models, jTrans (L_1) and Palmtree (L_2). The training cost for the remaining layers was much cheaper. Given the generalizability and the scalability of our approach, a one-time training is sufficient to achieve significant performance without adding further cost. Future work will include finding a replacement for jTrans and Palmtree at layers L_1 and L_2 to enable cross-architecture predictions with minimal impact on the overall accuracy of BinSimStack.

VII. Conclusion

We showed how to break code obfuscation, such as Flatten and Virtualize, and code optimization with state-of-the-art accuracy. We also demonstrated the effectiveness of our approach on real-world malware and vulnerability samples. Although the BERT-based jTrans outperformed other standalone models, it could not capture all the aspects of input data necessary for classification, and the parallel graph projections and training on the same data using GNNs helped improve the overall ensemble accuracy remarkably. Our paper also inspires a novel direction of research for BFSD, motivating exposure of unique subsets of functions while providing a coherent environment to allow models to participate in the same team with a single goal of BFSD instead of competing with each other.

REFERENCES

[1] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Seninja: A symbolic execution plugin for binary ninja," *SoftwareX*, vol. 20, p. 101219, 12 2022.

[2] H. Fang, Y. Wu, S. Wang, and Y. Huang, "Multi-stage binary code obfuscation using improved virtual machine," in *Information Security*, X. Lai, J. Zhou, and H. Li, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 168–181.

[3] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Comput. Surv.*, vol. 46, no. 1, Jul. 2013. [Online]. Available: https://doi.org/10.1145/2522968.2522972

[4] P. O'Kane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," *IEEE Security and Privacy*, vol. 9, no. 5, pp. 41–47, 2011.

[5] K. P. Grammatikakis, I. Koufos, N. Kolokotronis, C. Vassilakis, and S. Shiaeles, "Understanding and mitigating banking trojans: From zeus to emotet," in *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2021, pp. 121–128.

[6] A. Klopsch, "Attacking emotet's control flow flattening," 2022. [Online]. Available: https://news.sophos.com/en-us/2022/05/04/attacking-emotets-control-flow-flattening/

[7] G. Kucherin, "Combating control flow flattening in .net malware," 2022. [Online]. Available: https://www.virusbulletin.com/conference/vb2022/abstracts/combating-control-flow-flattening-net-malware/

[8] B. Levene, R. Falcone, and T. Halfpop, "Kazuar: Multiplatform espionage backdoor with api access," 2017. [Online]. Available: https://unit42.paloaltonetworks.com/unit42-kazuar-multiplatform-espionage-backdoor-api-access/

[9] N. Falliere, P. Fitzgerald, and E. Chien, "Inside the jaws of trojan. clampi," *Rapport technique, Symantec Corporation*, 2009.

[10] V. Hrčka, "Under the hood of wslink's multilayered virtual machine," in *Recon Conference*, 2022. [Online]. Available: https://www.welivesecurity.com/2022/03/28/under-hood-wslink-multilayered-virtual-machine/

[11] C. Collberg, S. Martin, J. Myers, B. Zimmerman, P. Krajca, G. Kerneis, S. Debray, and B. Yadegari, "The tigress c diversifier/obfuscator," 2015. [Online]. Available: https://tigress.wtf/index.html

[12] T. László and Á. Kiss, "Obfuscating c++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, no. 1, pp. 3–19, 2009.

[13] S. Udupa, S. Debray, and M. Madou, "Deobfuscation: reverse engineering obfuscated code," in *12th Working Conference on Reverse Engineering (WCRE'05)*, 2005, pp. 10 pp.–54.

[14] M. Sojer and J. Henkel, "Code reuse in open source software development: Quantitative evidence, drivers, and impediments," *Journal of the Association for Information Systems*, vol. 11, no. 12, pp. 868–901, 2010.

[15] A. Mockus, "Large-scale code reuse in open source software," in *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 7–7.

[16] B. Chandra, "A technical view of the openssl 'heartbleed' vulnerability," *IBM, Armonk, NY, USA, Tech. Rep.*, 2014.

[17] W. Stallings and L. Brown, *Computer Security: Principles and Practice*, 3rd ed. USA: Prentice Hall Press, 2014.

[18] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi, "Duqu: Analysis, detection, and lessons learned," in *ACM European Workshop on System Security (EuroSec)*, vol. 2012, 2012.

[19] J. Milletary, "Citadel trojan malware analysis," *Dell SecureWorks*, 2012.

[20] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jtrans: jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–13. [Online]. Available: https://doi.org/10.1145/3533767.3534367

[21] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3236–3251. [Online]. Available: https://doi.org/10.1145/3460120.3484587

[22] H. Wang, Z. Gao, C. Zhang, M. Sun, Y. Zhou, H. Qiu, and X. Xiao, "Cebin: A cost-effective framework for large-scale binary code similarity detection," ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 149–161. [Online]. Available: https://doi.org/10.1145/3650212.3652117

[23] H. Wang, Z. Gao, C. Zhang, Z. Sha, M. Sun, Y. Zhou, W. Zhu, W. Sun, H. Qiu, and X. Xiao, "Clap: Learning transferable binary code representations with natural language supervision," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 503–515. [Online]. Available: https://doi.org/10.1145/3650212.3652145

[24] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.

[25] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, 2019.

[26] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 480–491. [Online]. Available: https://doi.org/10.1145/2976749.2978370

[27] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*. PMLR, 2019, pp. 3835–3845.

[28] H. Huang, A. M. Youssef, and M. Debbabi, "Binsequence: Fast, accurate and scalable binary code reuse detection," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 155–166.

[29] L. Nouh, A. Rahimian, D. Mouheb, M. Debbabi, and A. Hanna, "Binsign: fingerprinting binary functions to support automated analysis of code executables," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2017, pp. 341–355.

[30] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 896–899.

[31] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2099–2116. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli

[32] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 667–678.

[33] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 472–489.

[34] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[35] L. Massarelli, G. A. D. Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 309–329.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[37] J. Devlin, M.-W. Chang, K. Lee, and K. 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)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[38] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent

variable models for structured data," in *International conference on machine learning*. PMLR, 2016, pp. 2702–2711.
[39] H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue, "Code is not natural language: Unlock the power of Semantics-Oriented graph representation for binary code similarity detection," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1759–1776. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/he-haojie
[40] A. Qasem, M. Debbabi, B. Lebel, and M. Kassouf, "Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures," ser. ASIA CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 443–456. [Online]. Available: https://doi.org/10.1145/3579856.3582818
[41] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *Network and Distributed System Security (NDSS) Symposium*, 02 2016.
[42] G. GCC, "Compiler optimization flags." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

APPENDIX A
TIGRESS USAGE

We used the Tigress source code obfuscator to obfuscate the code. We used the following Tigress commands for Control-flow Flattening, Virtualization, and cascading of Mixed-Boolean Arithmetic with Control-flow Flattening, respectively.

```
tigress
  --Environment=x86_64:Linux:Gcc:4.6
  --gcc=gcc
  --Transform=Flatten
  --Functions={fns} {lib_inc}
  --out={output_file}.c -c {input_file}

tigress
  --Environment=x86_64:Linux:Gcc:4.6
  --gcc=gcc
  --Transform=Virtualize
  --Functions={fns} {lib_inc}
  --out={output_file}.c -c {input_file}

tigress
  --Environment=x86_64:Linux:Gcc:4.6
  --gcc=gcc
  --Transform=EncodeArithmetic
  --Functions={fns}
  --Transform=Flatten
  --Functions={fns} {lib_inc}
  --out={output_file}.c -c {input_file}
```

In the above commands, `fns` refers to the functions that should be obfuscated, `lib_inc` refers to any library includes corresponding to the input C source code file to be linked, `output_file` refers to the name of the output file, and `input_file` refers to the input C source code file containing the functions to be obfuscated.

APPENDIX B
ENCODING OF CIRRINA

Cirrina aims to capture the branching structure of a function using tracelets at the node level (Fig. 12) and then exchanges this information among all the nodes during training. In machine learning, the application of *Graph Neural Networks* (GNN) to graph datasets is heavily biased towards using node and edge level semantics, and the construction of GNNs accordingly reflects reliance on node and edge level features as input for training. Depending on the nature of the graph dataset and the problem we are trying to solve, the nodes might carry more information than edges or vice versa. For example, to find the shortest path in a road transportation network or to solve a routing problem, the edges might carry more significant information than nodes. For solving a query search on the World Wide Web, the nodes might carry more information than edges. Sometimes, the tracelets can carry significant information compared to nodes and edges and, when used as standalone features, might help solve some problems efficiently. There has been limited usage of tracelets as input features for GNN, and in all those cases, the tracelets are not encoded as standalone features and are instead used to supplement the node or edge-level features. A graph dataset might exhibit learnable patterns at high-level semantics, such as the branching structure of a CFG as captured by Cirrina using tracelets, and we aim to exploit it by employing Cirrina to operate at layer L_4.

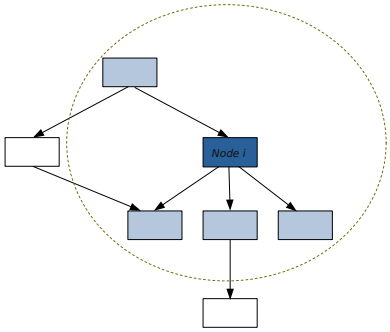
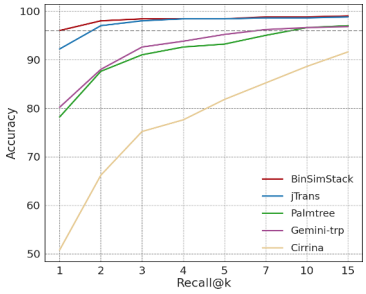


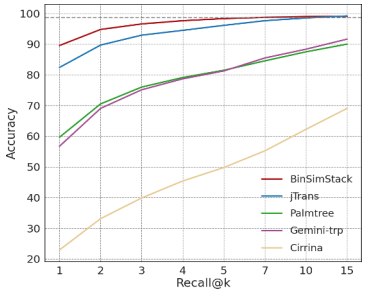
Fig. 12: Branching structure of a function at node i with one parent and three children

APPENDIX C
GENERALIZABILITY AND SCALABILITY

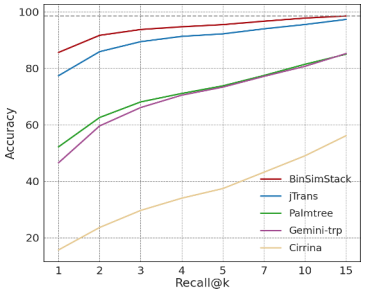
Here, we have given the scalability and generalizability analysis for Flatten, $O_0 \leftrightarrow O_1$, $O_0 \leftrightarrow O_2$, and $O_0 \leftrightarrow O_3$. Except for $O_0 \leftrightarrow O_3$, all the experiments follow the same settings as for Virtualize (Sec. V-C). For $O_0 \leftrightarrow O_3$, we used test sets of sizes 500, 2500, and 5000 due to insufficient (unique) samples. From Fig. 13,14,15,16, we can see that BinSimStack outperforms the participating models for all the transformations. Again, we can see from the figures that the gap between BinSimStack and jTrans (best-performing participant model) keeps increasing with the increase in the size of the dataset.



(a) Recall @ top-k (testset size = 500)

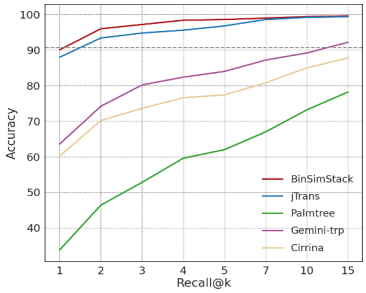


(b) Recall @ top-k (testset size = 3500)

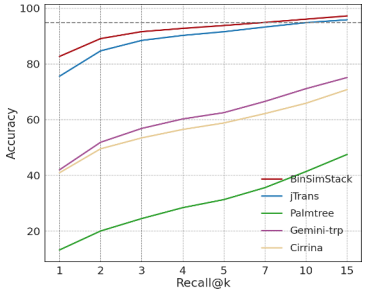


(c) Recall @ top-k (testset size = 7500)

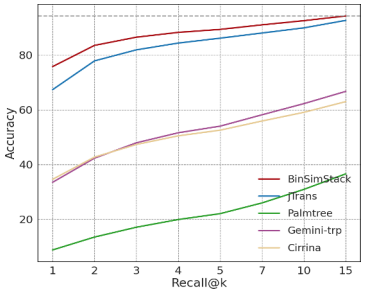
Fig. 13: Scalability and Generalizability analysis for Flatten



(a) Recall @ top-k (testset size = 500)

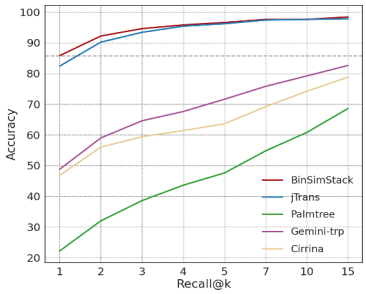


(b) Recall @ top-k (testset size = 3500)

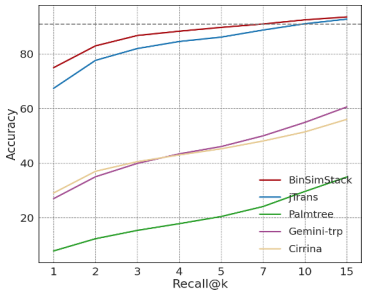


(c) Recall @ top-k (testset size = 7500)

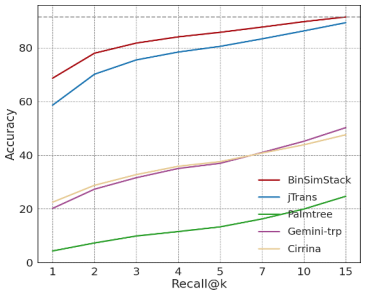
Fig. 14: Scalability and Generalizability analysis for $O_0 \leftrightarrow O_1$



(a) Recall @ top-k (testset size = 500)

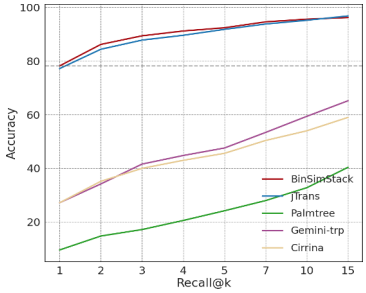


(b) Recall @ top-k (testset size = 3500)

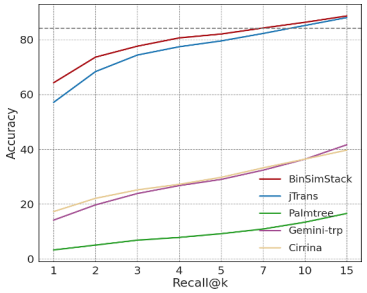


(c) Recall @ top-k (testset size = 7500)

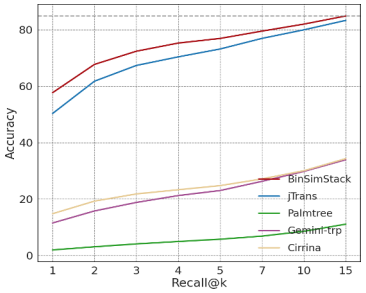
Fig. 15: Scalability and Generalizability analysis for $O_0 \leftrightarrow O_2$



(a) Recall @ top-k (testset size = 500)



(b) Recall @ top-k (testset size = 2500)



(c) Recall @ top-k (testset size = 5000)

Fig. 16: Scalability and Generalizability analysis for $O_0 \leftrightarrow O_3$