Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization

Steven H. H. Ding*, Benjamin C. M. Fung*, and Philippe Charland†
*Data Mining and Security Lab, School of Information Studies, McGill University, Montreal, Canada.
Emails: steven.h.ding@mail.mcgill.ca, ben.fung@mcgill.ca
†Mission Critical Cyber Security Section, Defence R&D Canada - Valcartier, Quebec, QC, Canada.
Email: philippe.charland@drdc-rddc.gc.ca

Abstract—**Reverse engineering is a manually intensive but necessary technique for understanding the inner workings of new malware, finding vulnerabilities in existing systems, and detecting patent infringements in released software. An assembly clone search engine facilitates the work of reverse engineers by identifying those duplicated or known parts. However, it is challenging to design a robust clone search engine, since there exist various compiler optimization options and code obfuscation techniques that make logically similar assembly functions appear to be very different.**

A practical clone search engine relies on a robust vector representation of assembly code. However, the existing clone search approaches, which rely on a manual feature engineering process to form a feature vector for an assembly function, fail to consider the relationships between features and identify those unique patterns that can statistically distinguish assembly functions. To address this problem, we propose to jointly learn the lexical semantic relationships and the vector representation of assembly functions based on assembly code. We have developed an assembly code representation learning model *Asm2Vec*. It only needs assembly code as input and does not require any prior knowledge such as the correct mapping between assembly functions. It can find and incorporate rich semantic relationships among tokens appearing in assembly code. We conduct extensive experiments and benchmark the learning model with state-of-the-art static and dynamic clone search approaches. We show that the learned representation is more robust and significantly outperforms existing methods against changes introduced by obfuscation and optimizations.

1. Introduction

Software developments mostly do not start from scratch. Due to the prevalent and commonly uncontrolled reuse of source code in the software development process [1], [2], [3], there exist a large number of clones in the underlying assembly code as well. An effective assembly clone search engine can significantly reduce the burden of the manual analysis process involved in reverse engineering. It addresses the information needs of a reverse engineer by taking advantage of existing massive binary data.

Assembly code clone search is emerging as an Information Retrieval (IR) technique that helps address security-related problems. It has been used for differing binaries to locate the changed parts [4], identifying known library functions such as encryption [5], searching for known programming bugs or zero-day vulnerabilities in existing software or Internet of Things (IoT) devices firmware [6], [7], as well as detecting software plagiarism or GNU license infringements when the source code is unavailable [8], [9]. However, designing an effective search engine is difficult, due to varieties of compiler optimizations and obfuscation techniques that make logically similar assembly functions appear to be dramatically different. Figure 1 shows an example. The optimized or obfuscated assembly function breaks control flow and basic block integrity. It is challenging to identify these semantically similar, but structurally and syntactically different assembly functions as clones.

Developing a clone search solution requires a robust vector representation of assembly code, by which one can measure the similarity between a query and the indexed functions. Based on the manually engineered features, relevant studies can be categorized into static or dynamic approaches. Dynamic approaches model the semantic similarity by dynamically analyzing the I/O behavior of assembly code [10], [11], [12], [13]. Static approaches model the similarity between assembly code by looking for their static differences with respect to the syntax or descriptive statistics [6], [7], [8], [14], [15], [16], [17], [18]. Static approaches are more scalable and provide better coverage than the dynamic approaches. Dynamic approaches are more robust against changes in syntax but less scalable. We identify two problems which can be mitigated to boost the semantic richness and robustness of static features. We show that by considering these two factors, a static approach can even achieve better performance than the state-of-the-art dynamic approaches.

P1: Existing state-of-the-art static approaches fail to consider the relationships among features. *LSH-S* [16], n-gram [8], n-perm [8], *BinClone* [15] and *Kam1n0* [17] model assembly code fragments as frequency values of operations and categorized operands. *Tracelet* [14] models assembly code as the editing distance between instruction sequences. *Discovre* [7] and *Genius* [6] construct descriptive features, such as the ratio of arithmetic assembly instructions, the number of transfer instructions, the number of basic blocks, among others. All these approaches assume each feature or category is an independent dimension. However, a *xmm0* Streaming SIMD Extensions (SSE) register is related to SSE operations such as *movaps*. A *fclose* libc function call is related to other file-related libc calls such as *fopen*. A *strcpy* libc call can be replaced with *memcpy*. These relationships provide more semantic information than

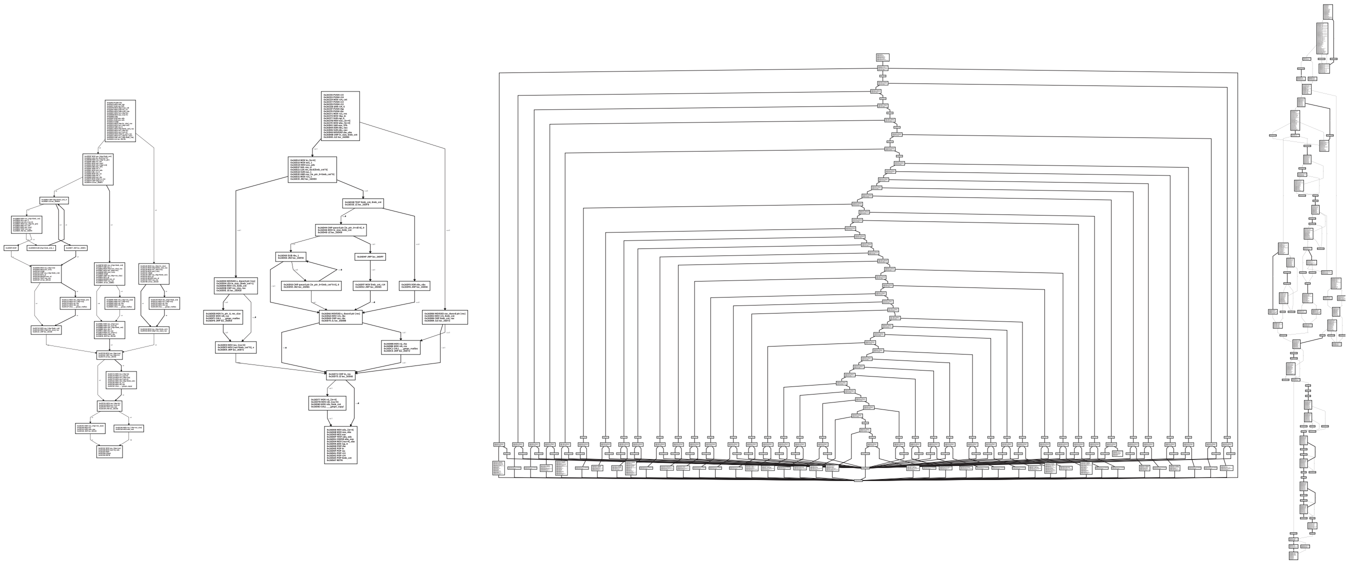


Figure 1: Different assembly functions compiled from the same source code of gmpz_tdiv_r_2exp in libgmp. From left to right, the assembly functions are compiled with gcc O0 option, gcc O3 option, LLVM obfuscator Control Flow Graph Flattening option, and LLVM obfuscator Bogus Control Flow Graph option. Asm2Vec can statically identify them as clones.

individual tokens or descriptive statistics.

To address this problem, we propose to incorporate lexical semantic relationship into the feature engineering process. Manually specifying all the potential relationships from prior knowledge of assembly language is time-consuming and infeasible in practice. Instead, we propose to learn these relationships directly from plain assembly code. *Asm2Vec* explores co-occurrence relationships among tokens and discovers rich lexical semantic relationships among tokens (see Figure 2). For example, *memcpy*, *strcpy*, *memncpy* and *mempcpy* appear to be semantically similar to each other. SSE registers relate to SSE operands. *Asm2Vec* does not require any prior knowledge in the training process.

P2: The existing static approaches assume that features are equally important [14], [15], [16], [17] or require a mapping of equivalent assembly functions to learn the weights [6], [7]. The chosen weights may not embrace the important patterns and diversity that distinguishes one assembly function from another. An experienced reverse engineer does not identify a known function by equally looking through the whole content or logic, but rather pinpoints critical spots and important patterns that identify a specific function based on past experience in binary analysis. One also does not need mappings of equivalent assembly code.

To solve this problem, we find that it is possible to simulate the way in which an experienced reverse engineer works. Inspired by recent development in representation learning [19], [20], we propose to train a neural network model to read many assembly code data and let the model identify the best representation that distinguishes one function from the rest. In this paper, we make the following contributions:

- We propose a novel approach for assembly clone detection. It is the first work that employs representation learning to construct a feature vector for assembly code, as a way to mitigate problems **P1** and **P2** in current handcrafted features. All previous research on assembly clone search requires a manual feature engineering process. The clone search engine is part of an open source platform[1].
- We develop a representation learning model, namely *Asm2Vec*, for assembly code syntax and control flow graph. The model learns latent lexical semantics between tokens and represents an assembly function as an internally weighted mixture of collective semantics. The learning process does not require any prior knowledge about assembly code, such as compiler optimization settings or the correct mapping between assembly functions. It only needs assembly code functions as inputs.
- We show that *Asm2Vec* is more resilient to code obfuscation and compiler optimizations than state-of-the-art static features and dynamic approaches. Our experiment covers different configurations of compiler and a strong obfuscator which substitutes instructions, splits basic blocks, adds bogus logics, and completely destroys the original control flow graph. We also conduct a vulnerability search case study on a publicly available vulnerability dataset, where *Asm2Vec* achieves zero false positives and 100% recalls. It outperforms a dynamic state-of-the-art vulnerability search method.

Asm2Vec as a static approach cannot completely defeat code obfuscation. However, it is more resilient to code obfuscation than state-of-the-art static features. This paper is organized as follows: Section 2 formally defines the search problem. Section 3 systematically integrates representation learning into a clone search process. Section 4 describes the model. Section 5 presents our experiment. Section 6 discusses the literature. Section 7 discusses the limitations and concludes the paper.

[1] https://github.com/McGill-DMaS/Kam1n0-Plugin-IDA-Pro

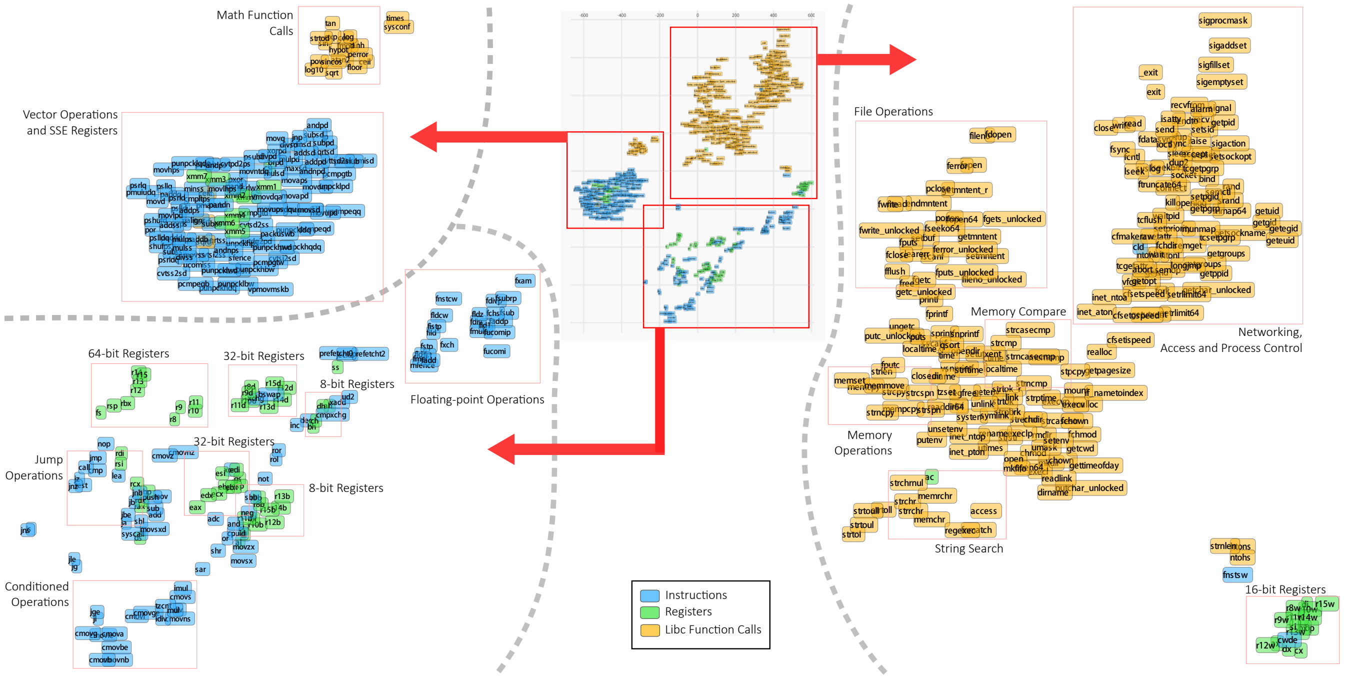


Figure 2: *T-SNE* clustering visualization of tokens appearing in assembly code. There are three categories of tokens: operation, operand, and *libc* function call. Each token is represented as a 200-dimensional numeric vector. They are learned by *Asm2Vec* on plain assembly code without any prior knowledge of the assembly language. The training assembly code does not contain the *libc* callee functions' content. For visualization, *T-SNE* reduces the vectors to two dimensions by nearest neighbor approximation. A smaller geometric distance indicates a higher lexical semantic similarity.

2. Problem Definition

In the assembly clone search literature, there are four types of clones [15], [16], [17]: Type I: literally identical; Type II: syntactically equivalent; Type III: slightly modified; and Type IV: semantically similar. We focus on Type IV clones, where assembly functions may appear syntactically different, but share similar functional logic in their source code. For example, the same source code with and without obfuscation, or a patched source code between different releases. We use the following notions: *function* denotes an assembly function; *source function* represents the original function written in source code, such as C++; *repository function* stands for the assembly function that is indexed inside the repository; and *target function* denotes the assembly function query. Given an assembly function, our goal is to search for its semantic clones from the repository RP. We formally define the search problem as follows:

Definition 1. *(Assembly function clone search) Given a target function* f_t*, the search problem is to retrieve the top-*k *repository functions* $f_s \in \text{RP}$*, ranked by their semantic similarity, so they can be considered as Type IV clones.*

3. Overall Workflow

Figure 3 shows the overall workflow. There are four steps: *Step 1:* Given a repository of assembly functions, we first build a neural network model for these functions. We only need their assembly code as training data without any prior knowledge. *Step 2:* After the training phase, the model produces a vector representation for each repository function. *Step 3:* Given a target function f_t that was not trained with this model, we use the model to estimate its vector representation. *Step 4:* We compare the vector of f_t against the other vectors in the repository by using cosine similarity to retrieve the top-k ranked candidates as results.

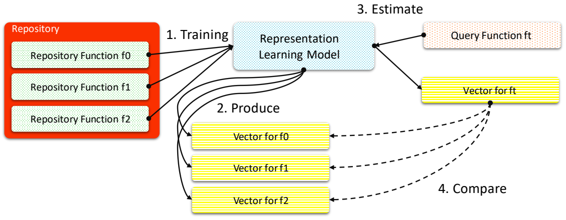


Figure 3: The overall work flow of *Asm2Vec*.

The training process is a one-time effort and is efficient to learn representation for queries. If a new assembly function is added to the repository, we follow the same procedure in Step 3 to estimate its vector representation. The model can be retrained periodically to guarantee the vectors' quality.

4. Assembly Code Representation Learning

In this section, we propose a representation learning model for assembly code. Specifically, our design is based on the *PV-DM* model [20]. *PV-DM* model learns document representation based on the tokens in the document. However, a document is sequentially laid out, which is different than assembly code, as the latter can be represented as a graph and has a specific syntax. First, we describe the original *PV-DM* neural network, which learns a vectorized representation of text paragraph. Then, we formulate our *Asm2Vec* model and describe how it is trained on instruction sequences for a given function. After, we elaborate how to model a control flow graph as multiple sequences.

4.1. Preliminaries

The *PV-DM* model is designed for text data. It is an extension of the original *word2vec* model. It can jointly learn vector representations for each word and each paragraph. Figure 4 shows its architecture.

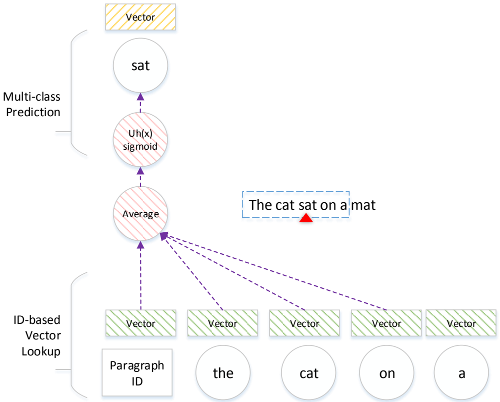


Figure 4: The PV-DM model.

Given a text paragraph which contains multiple sentences, *PV-DM* applies a sliding window over each sentence. The sliding window starts from the beginning of the sentence and moves forward a single word at each step. For example, in Figure 4, the sliding window has a size of 5. In the first step, the sliding window contains the five words 'the', 'cat', 'sat', 'on' and 'a'. The word 'sat' in the middle is treated as the *target* and the surrounding words are treated as the *context*. In the second step, the window moves forward a single word and contains 'cat', 'sat', 'on', 'a' and 'mat', where the word 'on' is the target.

At each step, the *PV-DM* model performs a multi-class prediction task (see Figure 4). It maps the current paragraph into a vector based on the paragraph ID and maps each word in the context into a vector based on the word ID. The model averages these vectors and predicts the target word from the vocabulary through a softmax classification. The back-propagated classification error will be used to update these vectors. Formally, given a text corpus T that contains a list of paragraphs $p \in \mathrm{T}$, each paragraph p contains a list of sentences $s \in p$, and each sentence is a sequence of $|s|$ words $w_t \in s$. *PV-DM* maximizes the log probability:

$$\sum_{p}^{T} \sum_{s}^{p} \sum_{t=k}^{|s|-k} log\ \mathbf{P}(w_t|p, w_{t-k}, ..., w_{t+k}) \qquad (1)$$

The sliding window size is $2k+1$. The paragraph vector captures the information that is missing from the context to predict the target. It is interpreted as topics [20]. *PV-DM* is designed for text data that is sequentially laid out. However, assembly code carries richer syntax than plaintext. It contains operations, operands, and control flow that are structurally different than plaintext. These differences require a different model architecture design that cannot be addressed by *PV-DM*. Next, we present a representation learning model that integrates the syntax of assembly code.

4.2. The Asm2Vec Model

An assembly function can be represented as a control flow graph (CFG). We propose to model the control flow graph as multiple sequences. Each sequence corresponds to a potential execution trace that contains linearly laid-out assembly instructions. Given a binary file, we use the IDA Pro[2] disassembler to extract a list of assembly functions, their basic blocks, and control flow graphs.

This section corresponds to Step 1 and 2 in Figure 3. In these steps, we train a representation model and produce a numeric vector for each repository function $f_s \in \mathrm{RP}$. Figure 5 shows the neural network structure of the model. It is different than the original *PV-DM* model.

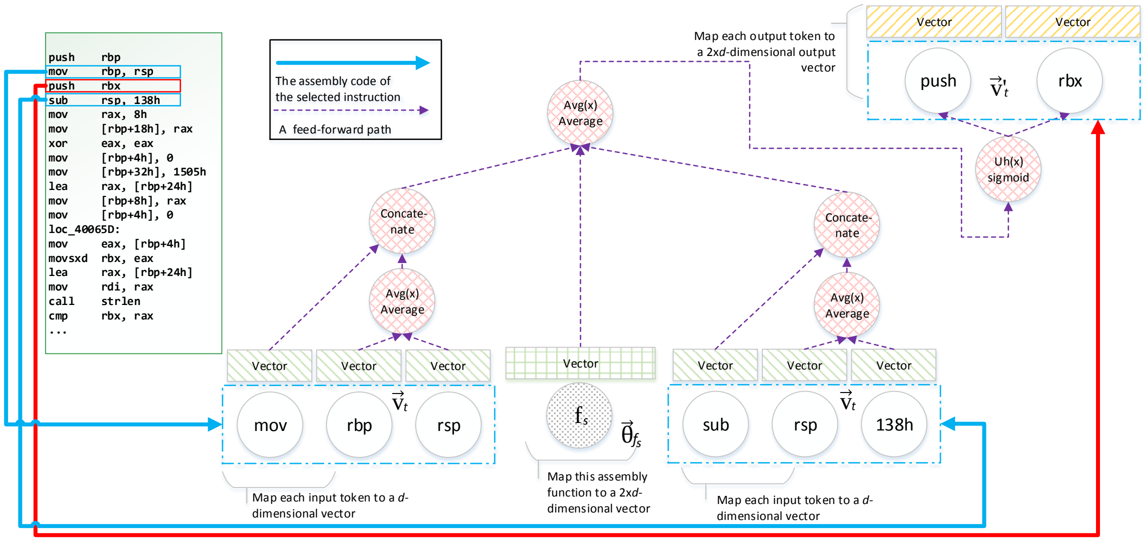
First, we map each repository function f_s to a vector $\vec{\theta}_{f_s} \in \mathbb{R}^{2\times d}$. $\vec{\theta}_{f_s}$ is the vector representation of function f_s to be learned in training. d is a user chosen parameter. Similarly, we collect all the unique tokens in the repository RP. We treat operands and operations in assembly code as tokens. We map each token t into a numeric vector $\vec{v}_t \in \mathbb{R}^d$ and another numeric vector $\vec{v'}_t \in \mathbb{R}^{2\times d}$. $\vec{v}_t$ is the vector representations of token t. After training, it represents a token's lexical semantics. $\vec{v}_t$ vectors are used in Figure 2 to visualize the relationship among tokens. $\vec{v'}_t$ is used for token prediction. All $\vec{\theta}_{f_s}$ and $\vec{v}_t$ are initialized to small random value around zero. All $\vec{v'}_t$ are initialized to zeros. We use $2\times d$ for f_s since we concatenate the vectors for operation and operands to represent an instruction.

We treat each repository function $f_s \in \mathrm{RP}$ as multiple sequences $\mathcal{S}(f_s) = seq[1:i]$, where seq_i is one of them. We assume that the order of sequences is randomized. A sequence is represented as a list of instructions $\mathcal{I}(seq_i) = in[1:j]$, where in_j is one of them. An instruction in_j contains a list of operands $\mathcal{A}(in_j)$ and one operation $\mathcal{P}(in_j)$. Their concatenation is denoted as its list of tokens $\mathcal{T}(in_j) = \mathcal{P}(in_j)\ ||\ \mathcal{A}(in_j)$, where $||$ denotes concatenation. Constants tokens are normalized into their hexadecimal form.

For each sequence seq_i in function f_s, the neural network walks through the instructions from its beginning. We collect the current instruction in_j, its previous instruction in_{j-1}, and its next instruction in_{j+1}. We ignore the instructions that are out-of-boundary. The proposed model tries to maximize the following log probability across the repository RP:

$$\sum_{f_s}^{\mathrm{RP}} \sum_{seq_i}^{\mathcal{S}(f_s)} \sum_{in_j}^{\mathcal{I}(seq_i)} \sum_{t_c}^{\mathcal{T}(in_j)} log\ \mathbf{P}(t_c|f_s, in_{j-1}, in_{j+1}) \qquad (2)$$

[2] IDA Pro, available at: http://www.hex-rays.com/



```
push    rbp
mov     rbp, rsp
push    rbx
sub     rsp, 138h
mov     rax, 8h
mov     [rbp+18h], rax
xor     eax, eax
mov     [rbp+4h], 0
mov     [rbp+32h], 1505h
lea     rax, [rbp+24h]
mov     [rbp+8h], rax
mov     [rbp+4h], 0
loc_40065D:
mov     eax, [rbp+4h]
movsxd  rbx, eax
lea     rax, [rbp+24h]
mov     rdi, rax
call    strlen
cmp     rbx, rax
...
```

The assembly code of the selected instruction
A feed-forward path
Map each output token to a 2xd-dimensional output vector
Vector
Vector
push
$\vec{v}'_t$
rbx
Uh(x) sigmoid
Avg(x) Average
Concate-nate
Avg(x) Average
Concate-nate
Avg(x) Average
Vector
Vector
Vector
Vector
Vector
Vector
Vector
mov
rbp
rsp
$\vec{v}_t$
f_s
$\vec{\theta}_{f_s}$
sub
rsp
138h
$\vec{v}_t$
Map each input token to a d-dimensional vector
Map this assembly function to a 2xd-dimensional vector
Map each input token to a d-dimensional vector

Figure 5: The proposed *Asm2Vec* neural network model for assembly code.

It maximizes the log probability of seeing a token t_c at the current instruction, given the current assembly function f_s and neighbor instructions. The intuition is to use the current function's vector and the context provided by the neighbor instructions to predict the current instruction. The vectors provided by neighbor instructions capture the lexical semantic relationship. The function's vector remembers what cannot be predicted given the context. It models the instructions that distinguish the current function from the others.

For a given function f_s, we first look-up its vector representation $\vec{\theta}_{f_s}$ through the previously built dictionary. To model a neighbor instruction in as $\mathcal{CT}(in) \in \mathbb{R}^{2\times d}$, we average the vector representations of its operands ($\in \mathbb{R}^d$) and concatenate the averaged vector ($\in \mathbb{R}^d$) with the vector representation of the operation. It can be formulated as:

$$\mathcal{CT}(in) = \vec{v}_{\mathcal{P}(in)} || \frac{1}{|\mathcal{A}(in)|} \sum_{t}^{\mathcal{A}(in)} \vec{v}_{t_b} \tag{3}$$

Recall that $\mathcal{P}(*)$ denotes an operation and it is a single token. By averaging f_s with $\mathcal{CT}(in_j - 1)$ and $\mathcal{CT}(in_j + 1)$, $\delta(in, f_s)$ models the joint memory of neighbor instructions:

$$\delta(in_j, f_s) = \frac{1}{3}(\vec{\theta}_{f_s} + \mathcal{CT}(in_{j-1}) + \mathcal{CT}(in_{j+1})) \tag{4}$$

Example 1. Consider a simple assembly code function f_s and one of its sequence in Figure 5. Take the third instruction where $j = 3$ for example. $\mathcal{T}(in_3)$ = {'push', 'rbx'}. $\mathcal{A}(in_{3-1})$ = {'rbp', 'rsp'}. $\mathcal{P}(in_{3-1})$ = {'mov'}. We collect their respective vectors $\vec{v}_{rbp}$, $\vec{v}_{rsp}$, $\vec{v}_{mov}$ and calculate $\mathcal{CT}(in_{3-1}) = \vec{v}_{mov} || (\vec{v}_{rbp} + \vec{v}_{rsp})/2$. Following the same procedure, we calculate $\mathcal{CT}(in_{3+1})$. With Equation 4 and $\vec{\theta}_{f_s}$ we have $\delta(in_3, f_s)$. ■

Given $\delta(in, f_s)$, the probability term in Equation 2 can be rewritten as follows:

$$\mathbf{P}(t_c | f_s, in_{j-1}, in_{j+1}) = \mathbf{P}(t_c | \delta(in_j, f_s)) \tag{5}$$

Recall that we map each token into two vectors $\vec{v}$ and $\vec{v'}$. For each target token $t_c \in \mathcal{T}(in_j)$, which belongs to the current instruction, we look-up its output vector $\vec{v'}_{t_c}$. The probability in Equation 5 can be modeled as a softmax multi-class regression problem:

$$\begin{aligned}
\mathbf{P}(t_c | \delta(in_j, f_s)) &= \mathbf{P}(\vec{v'}_{t_c} | \delta(in_j, f_s)) \\
&= \frac{f(\vec{v'}_{t_c}, \delta(in_j, f_s))}{\sum_d^D f(\vec{v'}_{t_d}, \delta(in_j, f_s))} \\
f(\vec{v'}_{t_c}, \delta(in_j, f_s)) &= Uh((\vec{v'}_{t_c})^T \times \delta(in_j, f_s))
\end{aligned}$$

D denotes the whole vocabulary constructed upon the repository RP. $Uh(\cdot)$ denotes a sigmoid function applied to each value of a vector. The total number of parameters to be estimated is $(|D| + 1) \times 2 \times d$ for each pass of the softmax layout. The term $|D|$ is too large for the softmax classification. Following [20], [21], we use the k negative sampling approach to approximate the log probability as:

$$\begin{aligned}
&log\ \mathbf{P}(t_c | \delta(in_j, f_s)) \approx log\ f(\vec{v'}_{t_c} | \delta(in_j, f_s)) \\
&+ \sum_{i=1}^{k} \mathbb{E}_{t_d \backsim P_n(t_c)} (\llbracket t_d \neq t_c \rrbracket log\ f(-1 \times \vec{v'}_{t_d},\ \delta(in_j, f_s)))
\end{aligned} \tag{6}$$

$\llbracket \cdot \rrbracket$ is an identity function. If the expression inside this function is evaluated to be true, then it outputs 1; otherwise 0. For example, $\llbracket 1 + 2 = 3 \rrbracket = 1$ and $\llbracket 1 + 1 = 3 \rrbracket = 0$. The negative sampling algorithm distinguishes the correct guess t_c with k randomly selected negative samples $\{t_d | t_d \neq t_c\}$ using $k + 1$ logistic regressions. $\mathbb{E}_{t_d \backsim P_n(t_c)}$ is a sampling function that samples a token t_d from the vocabulary D according to the noise distribution $P_n(t_c)$ constructed from D. By taking derivatives, respectively on $\vec{v'}_t$ and $\vec{\theta}_{f_s}$, we can calculate the gradients as follows.

$$
\begin{aligned}
\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta) &= \frac{1}{3} \sum_{i}^{k} \mathbb{E}_{t_b \backsim P_n(t_c)} \big([\![t_b = t_c]\!] - f(\vec{v'}_t, \delta(in_j, f_s)) \big) \\
&\quad \times \vec{v'}_t \\
\frac{\partial}{\partial \vec{v'}_t} J(\theta) &= [\![t = t_c]\!] - f(\vec{v'}_t, \delta(in_j, f_s)) \times \delta(in_j, f_s)
\end{aligned} \tag{7}
$$

By taking derivatives, respectively on $\vec{v}_{\mathcal{P}(in_{j+1})}$ and $\{\vec{v}_{t_b} | t_b \in \mathcal{A}(in_{j+1})\}$, we can calculate their gradients as follows. It will be the same equation for the previous instruction in_{j-1}, by replacing in_{j+1} with in_{j-1}.

$$
\begin{aligned}
\frac{\partial}{\partial \vec{v}_{\mathcal{P}(in_{j+1})}} J(\theta) &= \big(\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta) \big) [0 : d-1] \\
\frac{\partial}{\partial \vec{v}_{t_b}} J(\theta) &= \frac{1}{|\mathcal{A}(in_{j+1})|} \times \big(\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta) \big) [d : 2d-1] \\
& t_b \in \mathcal{A}(in_{j+1})
\end{aligned} \tag{8}
$$

After, we use back propagation to update the values of the involved vectors. Specifically, we update $\vec{\theta}_{f_s}$, all the involved $\vec{v}_t$ and involved $\vec{v'}_t$ according to their gradients, with a learning rate.

Example 2. Continue from Example 1, where the target token t_c is 'push'. Next, we calculate $\mathbf{P}(\vec{v'}_{push} | \delta(in_j, f_s))$ using negative sampling (Equation A). After, we calculate the gradients using Equation 7 and 8. We update all the involved vectors in these two examples, according to their respective gradient, with a learning rate. ■

4.3. Modeling Assembly Functions

In this section, we model an assembly function into multiple sequences. Formally, we treat each repository function $f_s \in \text{RP}$ as multiple sequences $\mathcal{S}(f_s) = seq[1 : i]$. The original linear layout of control flow graph covers some invalid execution paths. We cannot directly use it as a training sequence. Instead, we model the control flow graph as edge coverage sequences and random walks.

4.3.1. Selective Callee Expansion. Function inlining is a compiler optimization technique that replaces a function call instruction with the body of the called function. It extends the original assembly function and improves its performance by removing call overheads. It significantly modifies the control flow graph and is a major challenge in assembly clone search [12], [13].

BinGo [12] proposes to selectively inline callee functions into the caller function in the dynamic analysis process. We adopt this technique for static analysis. Function call instructions are selectively expanded with the body of the callee function. *BinGo* inlines all the standard library calls for the purpose of semantic correctness. We do not inline any library calls, since the lexical semantic among library call tokens have been well captured by the model (see the visualization in Figure 2). *BinGo* recursively inlines callee, but we only expand the first-order callees in the call graph. Expanding callee functions recursively will include too many callees' body into the caller, which makes the caller function statically more similar to the callee.

The decoupling metric used by *BinGo* captures the ratio of in-degree and out-degree of each callee function f_c:

$$
\alpha(f_c) = \text{outdegree}(f_c) / (\text{outdegree}(f_c) + \text{indegree}(f_c)) \tag{9}
$$

We adopt the same equation, as well as the same threshold value 0.01, to select a callee for expansion. Additionally, we find that if the callee function is longer than or has a comparable length to the caller, the callee will occupy a too large portion of the caller. The expanded function appears similar to the callee. Thus, we add an additional metric to filter out lengthy callees:

$$
\delta(f_s, f_c) = \text{length}(f_c) / \text{length}(f_s) \tag{10}
$$

We expand a callee if δ is less than 0.6 or f_s is shorter than 10 lines of instructions. The second condition is to accommodate wrapper functions.

4.3.2. Edge Coverage. To generate multiple sequences for an assembly function, we randomly sample all the edges from the callee-expanded control flow graph, until all the edges in the original graph are covered. For each sampled edge, we concatenate their assembly code to form a new sequence. This way, we ensure that the control flow graph is fully covered. The model can still produce similar sequences, even if the basic blocks in the control flow graph are split or merged.

4.3.3. Random Walk. *CACompare* [13] uses a random input sequence to analyze the I/O behavior of an assembly function. A random input simulates a random walk on the valid execution flow. Inspired by this method, we extend the assembly sequences for an assembly function by adding multiple random walks on the expanded control flow graph. This way, the generated sequence is much longer than the edge sampling.

Dominator is a widely used concept in control flow analysis and compiler optimizations. A basic block dominates another if one has to pass this block in order to reach the other. Multiple random walks will put a higher probability to cover basic block that dominate others. These popular blocks can be the indicator of loop structures or cover important branching conditions. Using random walks can be considered as a natural way to prioritize basic blocks that dominate others.

4.4. Tranining, Estimating and Searching

The training procedure corresponds to Algorithm 1. For each function in the repository, it generates sequences by edging sampling and random walks. For each sequence, it goes through each instruction and applies the *Asm2Vec* to update the vectors (Line 10 to 19). As shown in Algorithm 1,

Algorithm 1 Training the Asm2Vec model for one epoch

```
function TRAIN(Repository RP)
    shuffle(RP)
    for each $f_s \in \mathrm{RP}$ do
        for each $seq_i \in \mathcal{S}(f_s)$ do
            for $j = 1 \rightarrow (|seq_i| - 1)$ do
            ▷ Going through each instruction.
                lookup $f_s$'s representation $\vec{\theta}_{f_s}$
                calculate $\mathcal{CT}(in_{j-1})$ by Equ. 3
                calculate $\mathcal{CT}(in_{j+1})$ by Equ. 3
                calculate $\delta(in_j, f_s)$ by Equ. 4
                for each $tkn \in in_j$ do
                ▷ Going through each token
                    $targets \leftarrow \mathbb{E}_{t_b \backsim P_n(tkn)} \cup \{tkn\}$
                    ▷ Sample tokens from $P_n(tkn)$
                    calculate and cumulate gradient for $\vec{\theta}_{f_s}$ (Equ. 7)
                    calculate gradient for $\vec{v'}_t$ (Equ. 7)
                    update $\vec{v'}_t$
                    calculate and cumulate gradient for $in_{j-1}$ (Equ. 8)
                    calculate and cumulate gradient for $in_{j+1}$ (Equ. 8)
                update vectors for tokens of $in_{j-1}$
                update vectors for tokens of $in_{j+1}$
                update $\vec{\theta}_{f_s}$

function $\mathcal{S}$(Function $f_s$)
    $graph \leftarrow$ CFG($f_s$)
    $graph \leftarrow$ ExpandSellectiveCallee($graph$)
    $sequences \leftarrow \{\}$
    for each $edg \in$ SampleEdge($graph$) do
        $seq \leftarrow$ source($edg$) || target($edg$)
        ▷ Concatenate the source and the target blocks
        $sequences \leftarrow sequences \cup \{seq\}$
    for $i \leftarrow$ numRandomWalk do
        $seq \leftarrow$ RandomWalk($graph$)
        $sequences \leftarrow sequences \cup \{seq\}$
    return $sequences$
```

the training procedure does not require a ground-truth mapping between equivalent assembly functions.

The estimation step corresponds to Step 3 in Figure 3. For an unseen assembly function f_t as query $f_t \notin \mathrm{RP}$ that does not belong to the set of training assembly functions, we first associate it with a vector $\vec{\theta}_{f_t} \in \mathbb{R}^{2 \times d}$, which is initialized to a sequence of small values close to zero. Then, we follow the same procedure in the training process, where the neural network goes through each sequence of f_t and each instruction of the sequence. In every prediction step, we fix all $\vec{v}_t$ and $\vec{v'}_t$ in the trained model and only propagate errors to $\vec{\theta}_{f_t}$. At the end, we have $\vec{\theta}_{f_t}$ while the vectors for all $f_s \in \mathrm{RP}$ and $\{\vec{v}_t, \vec{v'}_t | t \in D\}$ remain the same. To search for a match, vectors are flattened and compared using cosine similarity.

Scalability is critical for binary clone search, as there may be millions of assembly functions inside a repository. It is practical to train *Asm2Vec* on a large-scale of assembly code. A similar model on text has been shown to be scalable to billions of text samples for training [21]. In this study, we only use pair-wise similarity for nearest neighbor searching. Pair-wise searching among low-dimensional fix-length vectors can be fast. In our experiment in Section 5.3, there are 139,936 functions. The average training time for each function is 49 milliseconds. The average query response time is less than 300 milliseconds.

Algorithm 2 Estimating a vector representation for a query

```
function ESTIMATE(Query Function $\mathrm{f_t}$)
    initialize $f_t$'s representation $\vec{\theta}_{f_t}$
    for each $seq_i \in \mathcal{S}(f_t)$ do
        for $j = 1 \rightarrow (|seq_i| - 1)$ do
            calculate $\mathcal{CT}(in_{j-1})$ by Equ. 3
            calculate $\mathcal{CT}(in_{j+1})$ by Equ. 3
            calculate $\delta(in_j, f_t)$ by Equ. 4
            for each $tkn \in in_j$ do
                $targets \leftarrow \mathbb{E}_{t_b \backsim P_n(tkn)} \cup \{tkn\}$
                calculate gradient for $\vec{\theta}_{f_t}$ (Equ. 7)
                update $\vec{\theta}_{f_t}$
```

5. Experiments

We compare *Asm2Vec* with existing available state-of-the-art dynamic and static assembly clone search approaches. All the experiments are conducted with an Intel Xeon 6 core 3.60GHz CPU with 32G memory. To simulate a similar environment in related studies, we limit the JVM to only 8 threads. There are four experiments. First, we benchmark the baselines against different compiler optimizations with *GCC*. Second, we evaluate clone search quality against different heavy code obfuscations with *CLANG* and *O-LLVM*. Third, we use all the binaries of the previous two. In the last one, we apply *Asm2Vec* on a publicly available vulnerability search dataset. All binary files are stripped before clone search. In all of the experiments, we choose $d = 200$, 25 negative samples, 10 random walks, and a decaying learning rate 0.025 for *Asm2Vec*. 200 corresponds to the suggested dimensionality ($2d$) used in [20].

5.1. Searching with Different Compiler Optimization Levels

In this experiment, we benchmark the clone search performance against different optimization levels with the *GCC* compiler version 5.4.0. We evaluate *Asm2Vec* based on 10 widely used utility and numeric calculation libraries in Table 1. They are chosen according to an internal statistic of the prevalence of FOSS libraries. We first compile a selected library using the *GCC* compiler with four different compiler optimization settings, which results in four different binaries. Then, we test every combination of two of them, which corresponds to two different optimization levels. Given two binaries from the same library but with different optimization levels, we link their assembly functions using the compiler-output debug symbols and generate a clone mapping between functions. This mapping is used as the ground-truth data for evaluation only. We search the first against the second in RP and after, we search for the second against the first in RP. Only the binary in the repository is used for training. We take the average of the two.

A higher optimization level contains all optimization strategies from the lower level. The comparison between O2 and O3 is the easiest one (Figure 6). On average, 26% bytes of a function are modified and none of the functions are identical. 40% of a control flow graph is modified and 65% function pairs share similar graph complexity. It can be considered as the best situation where the optimization

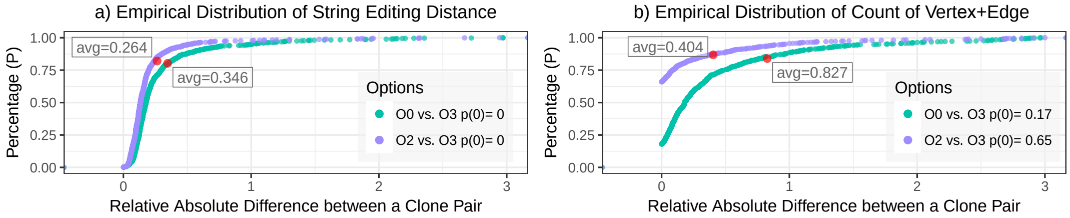


Figure 6: The difference between the O0/O2 optimized and the O3 optimized function. a) Relative string editing distance. 0.264 indicates that around 26.4% percent of bytes are different between two options for the same source code function. b) Relative absolute difference in the count of vertices and edges. 0.404% indicates that one function has 40.4% more vertices and edges than the other.

Compiler optimization O2 and O3

Baselines	BusyBox	CoreUtils	Libgmp	ImageMagick	Libcurl	LibTomCrypt	OpenSSL	SQLite	zlib	PuTTYgen	Avg.	p
BinGo†	Ø	.490	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.490	○
Composite	.789	.643	.910	.787	.842	.646	.783	.777	.813	.8IHI 38	.783	○
Constant	.437	.338	.202	.711	.522	.440	.365	.368	.549	.571	.450	●
Graphlet	.309	.268	.355	.262	.321	.297	.212	.313	.406	.148	.289	●
Graphlet-C	.662	.581	.680	.678	.689	.559	.586	.687	.730	.795	.665	●
Graphlet-E	.278	.225	.362	.270	.271	.219	.199	.280	.399	.355	.286	●
MixedGram	.811	.663	.906	.792	.848	.652	.789	.804	.858	.865	.798	●
MixedGraph	.445	.413	.436	.427	.486	.379	.350	.458	.564	.533	.449	●
n-gram	.774	.644	.874	.739	.814	.593	.748	.760	.812	.781	.754	●
n-perm	.803	.654	.912	.788	.848	.646	.785	.799	.850	.855	.793	●
FuncSimSearch	.157	.169	.848	.514	.663	.698	.726	.533	.488	.363	.516	●
PV(DM/DBOW)	.895	.899	.959	.952	.927	.945	.919	.898	.873	.823	.909	●
Asm2Vec*	**.954**	**.929**	**.973**	**.971**	**.951**	**.991**	**.931**	**.926**	**.885**	**.891**	**.940**	○

Compiler optimization O0 and O3

Baselines	BusyBox	CoreUtils	Libgmp	ImageMagick	Libcurl	LibTomCrypt	OpenSSL	SQLite	zlib	PuTTYgen	Avg.	p
BinGo†	Ø	.317	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.317	○
CACompare†	.844	Ø	Ø	**.893**	.794	Ø	**.795**	Ø	Ø	.717	.808	○
Composite	.013	.031	.019	.017	.004	.005	.007	.004	.036	.127	.026	●
Constant	.239	.128	.101	.610	.369	.258	.270	.182	.360	.439	.296	●
Graphlet	.017	.008	.049	.010	.023	.011	.009	.014	.029	.016	.019	●
Graphlet-C	.018	.020	.012	.022	.027	.001	.034	.012	.065	.102	.031	●
Graphlet-E	.021	.011	.075	.019	.017	.003	.018	.019	.051	.058	.029	●
MixedGram	.016	.033	.019	.018	.011	.005	.007	.006	.036	.116	.028	●
MixedGraph	.034	.028	.062	.024	.039	.015	.023	.030	.064	.097	.042	●
n-gram	.011	.029	.012	.021	.011	.010	.005	.003	.036	.129	.027	●
n-perm	.017	.029	.021	.021	.011	.006	.007	.005	.036	.129	.028	●
FuncSimSearch	.008	.019	.323	.039	.036	.030	.220	.011	.054	.040	.078	●
PV(DM/DBOW)	.745	.677	.760	.802	.792	.821	.759	.758	.713	.615	.744	●
Asm2Vec*	**.856**	**.781**	**.763**	.837	**.850**	**.921**	.792	**.776**	**.722**	**.788**	**.809**	○

TABLE 1: Clone search between different compiler optimization options using the *Precision at Position 1* (*Precision@1*) metric. It captures the ratio of assembly functions that are correctly matched at position 1. In this case, it equals *Recall at Position 1*. *Asm2Vec* is our proposed method. †denotes cited performance. ○ and ● respectively indicate $p > 0.05$ and $p \leq 0.01$ for Wilcoxon signed-rank test between *Asm2Vec* and each baseline.

strategies used in two binaries are similar. The comparison between O0 and O3 is the most difficult one. It can be considered as the worst situation where there exists a large difference in the optimization strategies (Figure 6). On average, 34% bytes of a function are modified and none of the functions are identical. 82% of a control flow graph is modified and 17% function pairs share similar graph complexity. Table 1 presents the results in these two situations. Due to the large number of cases, we only list the results for these two cases to demonstrate the best and worse situations. The results of other cases lie between these two and follow the same ranking.

Andriesse et al. [22] point out that using supervised machine learning may risk having invalid experiment results. For example, splitting *coreutils* binaries into training set and testing set may lead to an invalid good result since these binaries share a very similar code base. This issue is not applicable to our experiment. First, we follow the unsupervised learning paradigm, where the true clone mapping is only used for evaluation. Second, our training data is very different to the testing data, as shown in Figure 6 and Figure 7. For example, the *coreutils* library comes with many binaries but we statically linked them into a *single* binary. We train the O0-optimized binary and match the O3-optimized binary. These two binaries are very different.

We use the *Precision at Position 1* (*Precision@1*) metric. For every query, if a baseline returns no answer, we count the precision as zero. Therefore, *Precision@1* captures the ratio of assembly functions that are correctly matched, which is equal to *Recall at Position 1*. We benchmark nine feature representations proposed in [8]: mnemonic n-grams (denoted as n-gram), mnemonic n-perms (denoted as n-perm), Graphlets (denoted as *Graphlet*), Extended Graphlets (denoted as *Graphlet-E*), Colored Graphlets (denoted as *Graphlet-C*), Mixed Graphlets (denoted as *MixGraph*), Mixed n-grams/perms (denoted as *MixGram*), *Constants*, and the Composite of n-grams/perms and Graphlets (denoted as *Composite*). The idea of using *Graphlet* originated from [23]. These baseline methods cover a wide range of popular features from token to graph substructure. These baselines are configured according to the reported best settings in the paper. We also include the original *PV-DM* model and *PV-DBOW* model as a baseline where each assembly function is treated as a document. We pick the best results and denote it as *PV-(DM/DBOW)*. We only tune the configurations for *PV-(DM/DBOW)* as well as *Asm2Vec* on the *zlib* dataset. *FuncSimSearch* is an open source assembly clone static search toolkit recently released by Google[3]. It has a default training dataset that contains a ground-truth mapping of equivalent assembly functions. The state-of-the-art dynamic approach *BinGo* [12] and *CACompare* [13] are unavailable for evaluation. However, we conduct the experiment in the same way using the same metric. Their reported results are included in Table 1. We also include the Wilcoxon signed-rank test across different binaries to see if the difference in performance is statistically significant.

As shown in Table 1, *Asm2Vec* significantly outperforms static features in both the best and worse situation. It also outperforms *BinGo*, a recent semantic clone search approach that involves dynamic features. It shows that *Asm2Vec* is robust against heavy syntax modifications and intensive inlining introduced by the compiler. Even in the worse case, the learned representation can still correctly match more than 75% of assembly functions at position 1. It even achieves competitive performance against the state-of-the-art dynamic approach *CACompare* for semantic clone. The difference is not statistically different, due to the small sample size. *Asm2Vec* performs stably across different libraries and is able to find clones with high precision. On average, it achieves more than 93% precision in detecting clones among compiler optimization options O1, O2, and O3. As the difference between two optimization levels increases, the performance of the *Asm2Vec* decreases. Nevertheless, it is much less sensitive than the other static features, which demonstrates its robustness.

Discovre and *Genius* are two recent static approaches that use descriptive statistics and graph matching. Both of them are not available for evaluation. *CACompare* has been shown to outperform *Discovre* [7], *Genius* [6] and *Blanket* [10]. Our approach achieves comparable performance to *CACompare*, which indirectly compares *Asm2Vec*'s performance to *Discovre* and *Genius*.

In the best situation where we compare between optimization level O2 and O3, the baseline static features' performance is inline with the result reported in the original paper, which shows the correctness of our implementation. In the worse case, we notice that the *Constant* model outperforms the other static features based on assembly instructions and graph structures. The reason is that constant tokens do not suffer from changes in assembly instructions and subgraph structures. We also notice that *BinGo*, in the worse case, outperforms static features. However, in the best case, its performance is not as good as static features, such as *Graphlet-C* and n-grams, because the noise at the symbolic logic level is higher than at the assembly code level. Logical expressions promote recall and can find clones when the syntax is very different. However, assembly instructions can provide more precise information for matching.

The largest binary, OpenSSL, has more than 5,000 functions. *Asm2Vec* takes on average 153 ms to train an assembly function and 20 ms to process a query. For OpenSSL, *CACompare* takes on average 12 seconds to fulfill a query.

5.2. Searching with Code Obfuscation

Obfuscator-LLVM (O-LLVM) [24] is built upon the *LLVM* framework and the *CLANG* compiler toolchain. It operates at the intermediate language level and modifies a program's logics before the binary file is generated. It increases the complexity of the binary code. O-LLVM uses three different techniques and their combination: *Bogus Control Flow Graph (BCF)*, *Control Flow Flattening (FLA)*, and *Instruction Substitution (SUB)*. Figure 7 shows the statistics on differences.

- *BCF* modifies the control flow graph by adding a large number of irrelevant random basic blocks and branches. It will also split, merge, and reorder the original basic blocks. BCF breaks CFG and basic block integrity (on average 149% vertices/edges are added).
- *FLA* reorganizes the original CFG using complex hierarchy of new conditions as switches (see an example in Figure 1). The original instructions are heavily modified to accommodate the new entering conditions and variables. The linear layout has been completely modified (on average 376% vertices and edges are added). Graph-based features are oblivious to this technique. It is also unscalable for a dynamic approach to fully cover the CFG.
- *SUB* substitutes fragments of assembly code to its equivalent form by going one pass over the function logic using predefined rules. This technique modifies the contents of basic blocks and adds new constants. For example, additions are transformed to $a = b - (-c)$. Subtractions are transformed to $r = rand(); a = b-r; a = a-c; a = a+r$. And operations are transformed to $a = (b \wedge \backsim c) \& b$. *SUB* does not change much of the graph structure (91% of functions keep the same number of vertex and edge).
- *BCF+FLA+SUB* uses all the obfuscation options above.

O-LLVM heavily modifies the original assembly code. It breaks the CFG and the basic blocks integrity. By design, most of the static features are oblivious to the obfuscation. By using the *CLANG* compiler with *O-LLVM*, we successfully compile four libraries used in the last experiment and evaluate *Asm2Vec* using them. There were compilation errors when compiling the other binaries with the *CLANG+O-LLVM* toolchain. According to Figure 7, there is a significant difference between the original and the ones obfuscated with BCF and FLA. BCF doubles the number of vertices and edges. FLA almost doubles the latter. With SUB, the number

[3] Available at https://github.com/google/functionsimsearch

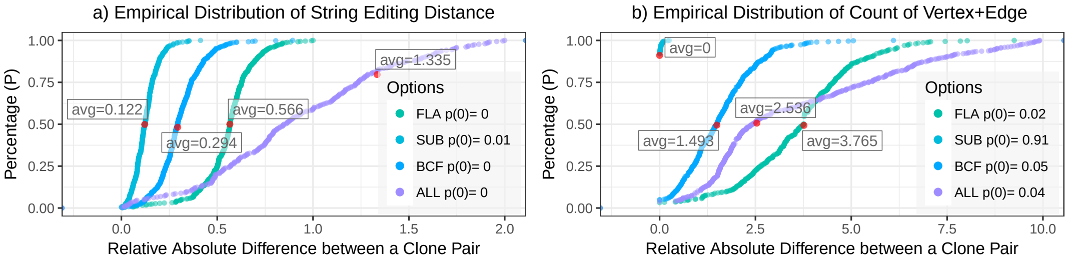


Figure 7: The difference between the original function and the obfuscated function. a) Relative string editing distance. 0.122 indicates that around 12.2% percent of bytes are modified. b) Relative absolute difference in the count of vertices and edge. 1.49% indicates the obfuscated function has 149% more vertices and edges in CFG.

	O-LLVM - Bogus Control Flow Graph (BCF)					O-LLVM - Instruction Substitution (SUB)				
Baselines	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.
Composite	.226	.224	.312	.246	.252	.620	.675	.600	.766	.665
Constant	.130	.592	.412	.318	.363	.173	.622	.492	.360	.412
Graphlet	.003	.005	.033	.007	.012	.198	.158	.411	.308	.269
Graphlet-C	.112	.118	.165	.124	.130	.626	.572	.539	.585	.581
Graphlet-E	.026	.011	.050	.014	.025	.454	.216	.286	.271	.307
MixedGram	.220	.234	.375	.303	.283	.585	.642	.563	.743	.633
MixedGraph	.011	.007	.049	.014	.020	.356	.325	.495	.488	.416
n-gram	.134	.134	.295	.195	.190	.466	.516	.513	.670	.541
n-perm	.233	.224	.374	.274	.276	.557	.624	.558	.736	.619
FuncSimSearch	.109	.022	.029	.027	.047	.685	.442	.699	.330	.539
PV(DM/DBOW)	.784	.870	.842	.768	.816	.935	.968	.964	.958	.956
Asm2Vec *	**.802**	**.920**	**.933**	**.883**	**.885**	**.940**	**.960**	**.981**	**.961**	**.961**

	O-LLVM - Control Flow Flattening (FLA)					O-LLVM - SUB+FLA+BCF				
Baselines	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.
Composite	.138	.129	.052	.027	.086	.219	.226	.015	.009	.117
Constant	.105	.480	.215	.209	.252	.137	.591	.173	.159	.265
Graphlet	.000	.002	.000	.000	.000	.000	.005	.000	.000	.001
Graphlet-C	.003	.008	.000	.001	.003	.107	.124	.000	.000	.058
Graphlet-E	.001	.002	.000	.000	.001	.020	.012	.000	.000	.008
MixedGram	.148	.143	.075	.036	.101	.221	.234	.018	.010	.121
MixedGraph	.003	.003	.000	.000	.002	.006	.010	.000	.000	.004
n-gram	.095	.093	.059	.030	.069	.154	.144	.013	.007	.079
n-perm	.133	.126	.055	.033	.087	.224	.222	.018	.008	.118
FuncSimSearch	.095	.001	.004	.008	.027	.110	.025	.003	.008	.037
PV(DM/DBOW)	**.852**	**.938**	.786	.763	.835	.780	.873	.639	.595	.722
Asm2Vec *	.772	.920	**.890**	**.795**	**.844**	**.854**	**.880**	**.830**	**.690**	**.814**

TABLE 2: Clone search between the original and obfuscated binary using the *Precision at Position 1* (*Precision@1*) metric. It captures the ratio of functions that are correctly matched at position 1, which is equal to *Recall at Position 1* (*Recall@1*) in this case. The difference between *Asm2Vec* and each baseline is significant (p <0.01 in a Wilcoxon signed-rank test).

of assembly instructions significantly increases. We use the same set of baselines and configurations from the previous experiment except for *BinGo* and *CACompare*, since they are unavailable for evaluation and the original papers do not include such an experiment.

We first compile a selected library without any obfuscation techniques applied. After, we compile the library again with a chosen obfuscation technique to have an original and an obfuscated binary. We link their assembly functions by using debug symbols and generate a one-to-one clone mapping between assembly functions. This mapping is used for evaluation purposes only. After stripping binaries, we search the original against the obfuscated. Then, we search for the obfuscated against the original. We report the average. We use the *Precision@1* as our evaluation measure. In this case, *Precision@1* equals *Recall@1*, since we treat 'no-answer' for a query as a zero precision.

Table 2 shows the results for *O-LLVM*. We find that instructions substitution can significantly reduce the performance of n-gram. SUB breaks the sequence by adding instructions in between. n-perm performs better than n-gram, since it ignores the order of tokens. Graph-based features can still recover more than 60% of clones, since the graph structure is not heavily modified. *Asm2Vec* can achieve more than 96% precision against assembly instruction substitution. Instructions are replaced with their equivalent form, which in fact still shares similar lexical semantic to the original. This information is well captured by *Asm2Vec*.

After applying BCF obfuscation, *Asm2Vec* can still achieve more than 88% precision, where the control flow graph already looks very different from the original. It shows that *Asm2Vec* is resilient to the inserted junk code and faked basic blocks. The FLA obfuscation destroys all the subgraph structures. This is also reflected from the degraded performance of graph sub-structure features. Most of them have a precision value around zero. Even in such situations, *Asm2Vec* can still correctly match 84% of assembly function clones. It shows that *Asm2Vec* is resilient to sub-structure

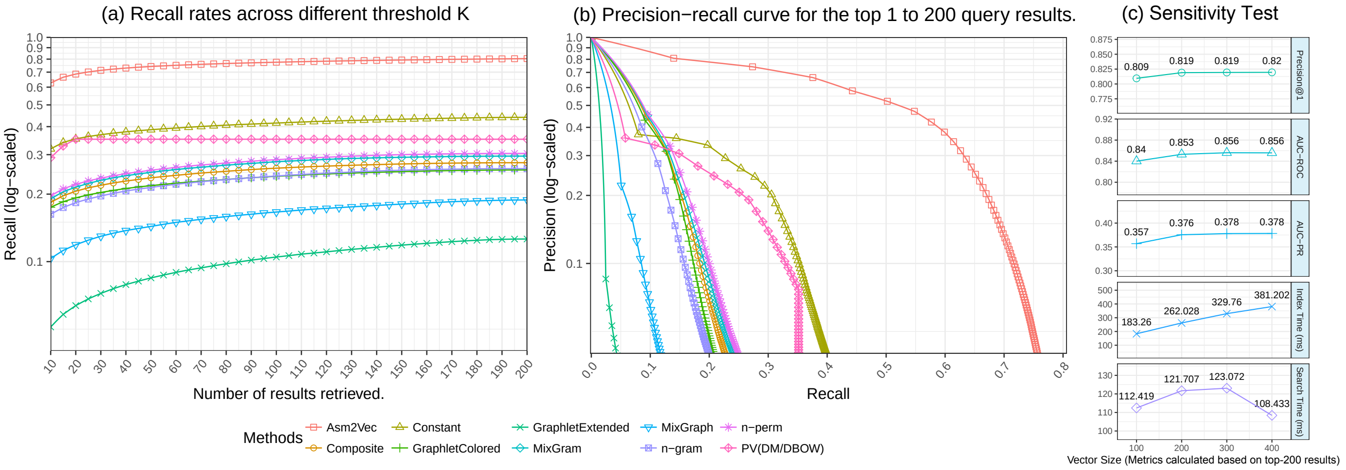


Figure 8: Baseline comparison for the third experiment. There are 139,936 assembly functions. We search each one against the rest. The set is a mixture of different compilers, compiler optimization settings, and *O-LLVM* obfuscation settings. a) Recall rates are plotted for different top-K retrieved results. b) Recall-Precision Curve. c) Sensitivity test on dimensionality.

changes and linear layout changes. After applying all the obfuscation techniques, *Asm2Vec* can still recover around 81% of assembly functions.

Asm2Vec can correctly pinpoint and identify critical patterns from noise. Inserted junk basic blocks or noise instructions follow the general syntax of random assembly code, which can be easily predicted by neighbor instructions. The function representation in *Asm2Vec* captures the missing information that cannot be provided by neighbor instructions. It also weights this information to best distinguish one function from another.

5.3. Searching against All Binaries

In this experiment, we use all the binaries in the previous two experiments. We evaluate whether *Asm2Vec* can distinguish different assembly functions when the candidate set is large. We also evaluate its performance with varying retrieval thresholds to inspect whether true positives are ranked at the top. Specifically, there are in total 60 binaries, which are a mixture of libraries compiled for different compiler options (O0-O3), different compilers (*GCC* and *CLANG*), and different *O-LLVM* obfuscation configurations. Following the experiment in *Genius* [6] and *Discovre* [7], we consider assembly functions that have at least 5 basic blocks. However, we do not use sampling. We use all of them. In total, there are 139,936 assembly functions. For each of them, we search against the rest to find clones. We sort the returned results and evaluate each of them in sequence. We use the same set of baselines and configuration from the last experiment except for *FuncSimSearch*, since it throws segmentation fault when indexing all the binaries.

We collect recall and precision at different top-k positions. We plot recall against k in Figure 8(a). We remove *Graphlet* from the figure, since it does not perform any better than *Graphlet-Extended*. Even with a large size of assembly functions, *Asm2Vec* can still achieve a recall of 70% for the top 20 results. It significantly outperforms other traditional token-based and graph-based features. Moreover, we observe that token-based approaches in general perform better than subgraph-based approaches.

		ESH [18]			Asm2Vec		
Vulnerability	CVE	FP	ROC	CROC	FP	ROC	CROC
Heartbleed	2014-0160	0	1	1	0	1	1
Shellshock	2014-6271	3	0.999	0.996	0	1	1
Venom	2015-3456	0	1	1	0	1	1
Clobberin' Time	2014-9295	19	0.993	0.956	0	1	1
Shellshock #2	2014-7169	0	1	1	0	1	1
ws-snmp	2011-0444	1	1	0.997	0	1	1
wget	2014-4877	0	1	1	0	1	1
ffmpeg	2015-6826	0	1	1	0	1	1

TABLE 3: Evaluating *Asm2Vec* on the vulnerability dataset [18] using the False Positives (FP), Receiver Operating Characteristic (ROC), and Concentrated ROC (CROC) metrics. For all the cases, *Asm2Vec* retrieves all results without any false positives.

We plot precision against recall for each baseline in Figure 8(b). This curve evaluates a clone search engine with respect to the trade-off between precision and recall, when varying the number of retrieved results. As shown in the plot, *Asm2Vec* outperforms traditional representations of assembly code. It achieves 82% precision for the returned top clone search result where $k = 1$. The false positives on average have 33 basic blocks ($\sigma = 231$). On the other hand, all the functions in the dataset on average have 47 basic blocks ($\sigma = 110$) as a prior. By using a one-sided Kolmogorov-Smirnov test, we can conclude that false positives have a smaller number of basic blocks than the overall population ($p < 2.2e^{-16}$). We conduct a sensitivity test based on top-200 results to evaluate different choices of vector size. Figure 8 (c) shows that with difference vector size *Asm2Vec* is stable for both efficacy and efficiency. We tried to incorporate more neighbor instructions. However, this increases the possible patterns to be learned and requires more data. In our experiment, we did not find such design effective.

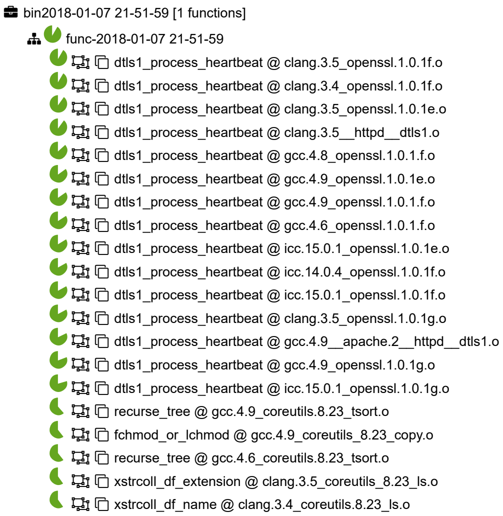
5.4. Searching Vulnerability Functions

In the above experiments, we evaluate *Asm2Vec*'s overall performance on matching general assembly functions. In this case study, we apply *Asm2Vec* on a publicly available vulnerability dataset[4] presented in [18] to evaluate its performance in actually recovering the reuse of the vulnerabilities in functions. The dataset contains 3,015 assembly functions. For each of the 8 given vulnerabilities, the task is to retrieve its variants from the dataset. The variants are either from different source code versions or generated by different versions of GCC, ICC and CLANG compilers. This dataset is closely related to the real-life scenario.

Figure 9 shows an example of using *Asm2Vec* to search for the *Heartbleed* vulnerability in the dataset. The query is a function containing the *Heartbleed* vulnerability in *OpenSSL* version 1.0.1f, compiled with Clang 3.5. There are total 15 different functions containing this vulnerability. The pie chart in each ranked entry indicates the similarity. Each ranked entry contains the assembly function name and its corresponding binary file. As shown in the ranked list, *Asm2Vec* successfully retrieves all the 15 candidates in the top 15 results. Therefore, it has a precision and recall of 1 for this query. The first entry corresponds to the same function as the query. However, it does not have a similarity of 1 since the query's representation is estimated but the one in repository is trained. However, it is still ranked first.

We implement *Asm2Vec* as an open source vulnerability search engine and follow the same experimental protocol to compare its performance with the state-of-the-art vulnerability search solution in [18]. Table 3 shows the results. We use the same performance metrics as [18]: False Positives (FP), Receiver Operating Characteristic (ROC), and Concentrated ROC (CROC). For all the vulnerabilities, *Asm2Vec* has zero false positives and 100% recall. Therefore, it achieves a ROC and a CROC of 1. It outperforms [18].

Tigress [25] is another advanced obfuscator. It transforms the C Intermediate Language (CIL) using virtualization and Just-In-Time (JIT) execution. *Tigress* failed to obfuscate a complete library binary due to compilation errors. Therefore we were unable to evaluate *Asm2Vec* against *Tigress* in the same way as against *O-LLVM* in Section 5.2. We increase the difficulty on the vulnerability search by using the *Tigress* obfuscator. In this experiment, for each of the 8 different vulnerabilities, we obfuscate the query function with literals encoded, virtualization, and Just-In-Time execution. Then, we try to recover their original variants from the dataset. *Encode Literals*: Literal integers are replaced with opaque expressions. Literal strings are replaced with a function that generates them at runtime. *Virtualization*: This transformation turns a function into an interpreter with specialized byte code translation. By design, it is difficult for a static approach to detect clones protected by this technique. *JIT*: It transforms the function to generate its code at runtime. Almost every instruction is replaced with a function call. By design, a static approach can hardly recover any variants. Our result shows that *Asm2Vec* is still able to recover 97.2% with literals encoded, 35% with virualization, and 45% with JIT execution (see Table 4). We inspect the result and find that *Asm2Vec* tries to match any similar information neglected by the obfuscator. However, after applying three obfuscation techniques at the same time, *Asm2Vec* can no longer recover any clone.



```
bin2018-01-07 21-51-59 [1 functions]
    func-2018-01-07 21-51-59
        dtls1_process_heartbeat @ clang.3.5_openssl.1.0.1f.o
        dtls1_process_heartbeat @ clang.3.4_openssl.1.0.1f.o
        dtls1_process_heartbeat @ clang.3.5_openssl.1.0.1e.o
        dtls1_process_heartbeat @ clang.3.5__httpd__dtls1.o
        dtls1_process_heartbeat @ gcc.4.8_openssl.1.0.1.f.o
        dtls1_process_heartbeat @ gcc.4.9_openssl.1.0.1e.o
        dtls1_process_heartbeat @ gcc.4.9_openssl.1.0.1.f.o
        dtls1_process_heartbeat @ gcc.4.6_openssl.1.0.1.f.o
        dtls1_process_heartbeat @ icc.15.0.1_openssl.1.0.1e.o
        dtls1_process_heartbeat @ icc.14.0.4_openssl.1.0.1f.o
        dtls1_process_heartbeat @ icc.15.0.1_openssl.1.0.1f.o
        dtls1_process_heartbeat @ clang.3.5_openssl.1.0.1g.o
        dtls1_process_heartbeat @ gcc.4.9__apache.2__httpd__dtls1.o
        dtls1_process_heartbeat @ gcc.4.9_openssl.1.0.1g.o
        dtls1_process_heartbeat @ icc.15.0.1_openssl.1.0.1g.o
        recurse_tree @ gcc.4.9_coreutils.8.23_tsort.o
        fchmod_or_lchmod @ gcc.4.9_coreutils_8.23_copy.o
        recurse_tree @ gcc.4.6_coreutils.8.23_tsort.o
        xstrcoll_df_extension @ clang.3.5_coreutils_8.23_ls.o
        xstrcoll_df_name @ clang.3.4_coreutils.8.23_ls.o
```

Figure 9: Searching the Heartbleed vulnerable function in the vulnerability dataset. The binary name indicates the compiler, library name, and library version. For example, clang.3.5_openssl.1.0.1f indicates that the binary is library *OpenSSL* version 1.0.1f compiled with *clang* version 3.5.

6. Related Work

Static approaches such as *k-gram* [26], *LSH-S* [16], n-gram [8], *BinClone* [15], *ILine* [27], and *Kam1n0* [17] rely on operations or categorized operands as static features. *BinSequence* [28] and *Tracelet* [14] model assembly code as the editing distance between instruction sequences. All these features failed to leverage the semantic relationship between operations or categories. *TEDEM* [29] compares basic blocks by their expression trees. However, even semantically similar instructions result in different expressions and side effects, which make them sensitive to instruction changes. *ILine* [27], *Discovre* [7], *Genius* [6], *BinSign* [30], and *BinShape* [31] construct descriptive statistic features, such as ratio of arithmetic assembly instructions, ratio of transfer instructions, number of basic blocks, and number of function calls, among others. Instruction-based features failed to consider the relationships between instructions and are affected by instruction substitutions. In NLP tasks one usually penalizes frequent words by filtering, subsampling or generalization. For assembly language we find that frequent words improve the robustness of the representation. Graph-based features are oblivious to CFG manipulations. *BinDiff* [32] and *BinSlayer* [33] rely on CFG matching, which is susceptible to CFG changes such as flattening. *Gitz* [34] is another static approach that used at the IR level. However, it operates at the boundary of a basic block and

[4] Available at https://github.com/nimrodpar/esh-dataset-1523

Searching with Obfuscation Options in Tigress									
Name	Heartbleed	ShellshocK	Venom	Clobberin' Time	Shellshock #2	ws-snmp	wget	ffmpeg	avg.
CVE	2014-0160	2014-6271	2015-3456	2014-9295	2014-7169	2014-4877	2014-4877	2015-6826	
# of Positives (k)	15	9	6	10	3	7	3	7	
Encode Literal	100%	77.8%	100%	100%	100%	100%	100%	100%	97.2%
Virtualization	0%	0%	100%	20%	100%	0%	66.7%	0%	35.8%
JIT Execution	53.3%	0%	83.3%	30%	33.3%	0%	0%	100%	37.5%

TABLE 4: True Positive Rate (TPR) of the top-k results searching the obfuscated vulnerable function against the dataset in [18]. k is chosen as the number of ground-truth clones in the dataset. For example, *Venom CVE 2015-3456-4877* has 6 variants in the dataset. By inspecting the top-6 results from *Asm2Vec* we recovered 100% (6/6) for the query with literals encoded, 100% (6/6) for the virtualized query, and 83.3% (5/6) for JIT-transformed query. After applying all the options at the same time, *Asm2Vec* cannot recover any true positives.

assumes basic block integrity, which is vulnerable to splitting. [35] proposes a graph convolution approach. It might be able to mitigate graph manipulation. However, it relies on supervised learning and requires a ground-truth mapping of equivalent assembly functions to be trained. *Asm2Vec* enriches static features by considering the lexical semantic relationships between tokens appearing in assembly code. It also avoids direct use of the graph-based features and is more robust against CFG manipulations. However, the CFG is useful in some malware analysis scenarios, especially for matching template-generated and marco-generated functions that share similar CFG structure. One direction is to combine *Asm2Vec* and *Tracelet* [14] or subgraph search [17].

Dynamic methods measure semantic similarity by dynamically analyzing the behavior of the target assembly code. *BinHunt* [36], *iBinHunt* [37], and *ESH* [18] use a theorem prover to verify whether two basic blocks or strands are equivalent. *BinHunt* and *iBinHunt* assume basic blocks integrity. *ESH* assumes strand integrity. They are vulnerable to block splitting. Jiang et al. [38], *Blex* [10], *Multi-MH* [11], and *BinGo* [12] use randomly-sampled values to compare I/O values. Random sampling may not correctly discriminate two logics. Consider that one expression outputs 1 if $v! = 100$; otherwise, 0. Another expression outputs 1 if $v! = 20$, otherwise, 0. Given a widely-used sampling range $[-1000, 1000]$, they have a high chance of being equivalent. *CACompare* follows the similar idea used in [39], [40], [41]. Besides of I/O values, it records all intermediate execution results and library function calls for matching. Using similar experiments to match assembly functions, *CACompare* achieves the best performance among the binary clone search literature at the time of writing this paper. However, it depends on a single input value and only covers one execution path. As stated by the authors, it is vulnerable to CFG changes. *Asm2Vec* leverages the lexical semantic rather than the symbolic relationship which is more scalable and less vulnerable to added noisy logics. As a static approach, *Asm2Vec* achieves competitive performance compared to *CACompare*. *CryptoHunt* is a recent dynamic approach for matching cryptographic functions. It can detect wrapped cryptographic API calls. *Asm2Vec* focuses on assembly code similarity, which is different to *CryptoHunt*.

Source code clone is another related area. *CCFINDERX* [42] and *CP-Miner* [43] use lexical tokens as features to find code clones. Baxter et al. [44] and *Deckard* [45] leverage abstract syntax trees for clone detection. *ReDeBug* [46] is another scalable source code search engine. Recently, deep learning has been applied on this problem [47].

7. Limitations and Conclusion

Asm2Vec suffers from several limitations. First, it is designed for a single assembly code language and the clone search engine is architecture-agnostic. At this stage, it is not directly applicable for semantic clones across architectures. In the future, we will align the lexical semantic space between two different assembly languages by considering their shared tokens, such as constants and libc calls. Second, the current selective callee expansion mechanism cannot determine the dynamic jumps, such as jump table. Third, as a black box static approach, *Asm2Vec* cannot explain or justify the returned results by showing the cloned subgraphs or proving symbolic equivalence. It has limited interpretability.

In this paper, we propose a robust and accurate assembly clone search approach named *Asm2Vec*, which learns a vector representation of an assembly function by discriminating it from the others. *Asm2Vec* does not require any prior knowledge such as the correct mapping between assembly functions or the compiler optimization level used. It learns lexical semantic relationships of tokens appearing in assembly code, and represents an assembly function as an internally weighted mixture of latent semantics. Besides assembly functions, it can be applied on different granularities of assembly sequences, such as binaries, fragments, basic blocks, or functions. We conduct extensive experiments on assembly code clone search, using different compiler optimization options and obfuscation techniques. Our results suggest that *Asm2Vec* is accurate and robust against severe changes in the assembly instructions and control flow graph.

Acknowledgments

The authors would like to thank the reviewers for the thorough reviews and valuable comments. This research is supported by Defence Research and Development Canada (contract no. W7701-155902/001/QCL), NSERC Discovery Grants (RGPIN-2018-03872), and Canada Research Chairs Program (950-230623).

References

[1] A. Mockus, “Large-scale code reuse in open source software,” in *Proceedings of the International Workshop on Emerging Trends in FLOSS Research and Development*. IEEE, 2007.

[2] 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, 2010.

[3] E. Juergens *et al.*, “Why and how to control cloning in software artifacts,” *Technische Universität München*, 2011.

[4] S. Brown. (2016) Binary diffing with kam1n0. [Online]. Available: https://www.whitehatters.academy/diffing-with-kam1n0/

[5] J. Qiu, X. Su, and P. Ma, “Library functions identification in binary code by using graph isomorphism testings,” in *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2015.

[6] 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*, 2016.

[7] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, “discovre: Efficient cross-architecture identification of bugs in binary code,” in *Proceedings of the 23rd Symposium on Network and Distributed System Security (NDSS)*, 2016.

[8] W. M. Khoo, A. Mycroft, and R. J. Anderson, “Rendezvous: a search engine for binary code,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013.

[9] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, “Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[10] M. Egele, M. Woo, P. Chapman, and D. Brumley, “Blanket execution: Dynamic similarity testing for program binaries and components,” in *Proceedings of the 23rd USENIX conference on Security*, 2014.

[11] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, “Cross-architecture bug search in binary executables,” in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015.

[12] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, “Bingo: cross-architecture cross-os binary search,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[13] Y. Hu, Y. Zhang, J. Li, and D. Gu, “Binary code clone detection across architectures and compiling configurations,” in *Proceedings of the 25th International Conference on Program Comprehension*, 2017.

[14] Y. David and E. Yahav, “Tracelet-based code search in executables,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[15] M. R. Farhadi, B. C. M. Fung, P. Charland, and M. Debbabi, “Binclone: Detecting code clones in malware,” in *Proceedings of the 8th International Conference on Software Security and Reliability*, 2014.

[16] A. Saebjornsen, “Detecting fine-grained similarity in binaries,” Ph.D. dissertation, UC Davis, 2014.

[17] S. H. H. Ding, B. C. M. Fung, and P. Charland, “Kam1n0: Mapreduce-based assembly clone search for reverse engineering,” in *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*.

[18] Y. David, N. Partush, and E. Yahav, “Statistical similarity of binaries,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

[19] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[20] Q. Le and T. Mikolov, “Distributed representations of sentences and documents,” in *Proceedings of the International Conference on Machine Learning*, 2014, pp. 1188–1196.

[21] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Proceedings of the Advances in Neural Information Processing Systems*, 2013.

[22] D. Andriesse, A. Slowinska, and H. Bos, “Compiler-agnostic function detection in binaries,” in *Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017, pp. 177–189.

[23] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, “Polymorphic worm detection using structural information of executables,” in *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*. Springer, 2006.

[24] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, “Obfuscator-llvm–software protection for the masses,” in *Proceedings of 2015 IEEE/ACM 1st International Workshop on Software Protection (SPRO)*. IEEE, 2015.

[25] S. Banescu, C. S. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, “Code obfuscation against symbolic execution attacks,” in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, 2016, pp. 189–200.

[26] G. Myles and C. Collberg, “K-gram based software birthmarks,” in *Proceedings of the 2005 ACM Symposium on Applied Computing*. ACM, 2005, pp. 314–318.

[27] J. Jang, M. Woo, and D. Brumley, “Towards automatic software lineage inference.” in *Proceedings of the USENIX Security Symposium*, 2013, pp. 81–96.

[28] H. Huang, A. Youssef, and M. Debbabi, “BinSequence: Fast, accurate and scalable binary code reuse detection.” in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*. ACM Press, 2017.

[29] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, “Leveraging semantic signatures for bug search in binary programs,” in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014.

[30] L. Nouh, A. Rahimian, D. Mouheb, M. Debbabi, and A. Hanna, “BinSign: Fingerprinting binary functions to support automated analysis of code executables,” in *Proceedings of the IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2017.

[31] P. Shirani, L. Wang, and M. Debbabi, “BinShape: Scalable and robust binary library function identification using function shape,” in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017.

[32] T. Dullien and R. Rolles, “Graph-based comparison of executable objects (english version),” *SSTIC*, vol. 5, no. 1, p. 3, 2005.

[33] M. Bourquin, A. King, and E. Robbins, “Binslayer: accurate comparison of binary executables,” in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 2013, p. 4.

[34] Y. David, N. Partush, and E. Yahav, “Similarity of binaries through re-optimization,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017, pp. 79–94.

[35] 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*. ACM, 2017, pp. 363–376.

[36] D. Gao, M. K. Reiter, and D. Song, “Binhunt: Automatically finding semantic differences in binary programs,” in *Proceedings of the International Conference on Information and Communications Security*. Springer, 2008.

[37] J. Ming, M. Pan, and D. Gao, "ibinhunt: Binary hunting with inter-procedural control flow," in *Proceedings of the International Conference on Information Security and Cryptology*. Springer, 2012.

[38] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*. ACM, 2009.

[39] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011.

[40] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012.

[41] X. Zhang and R. Gupta, "Matching execution histories of program versions," in *Proceedings of the 10th European Software Engineering Conference*. ACM, 2005.

[42] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, 2002.

[43] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, 2006.

[44] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of International Conference on Software Maintenance*. IEEE, 1998.

[45] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007.

[46] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *Proceedings of IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012.

[47] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016.

Appendix A.
Extended Formulation of Asm2Vec

This appendix extends the original description and the formulation of the *Asm2Vec* model training. Recall that we define f_s as an assembly function in the repository at the beginning of Section 4. The *Asm2Vec* model tries to learn the following parameters:

$\vec{\theta}_{f_s} \in \mathbb{R}^{2\times d}$	The vector representation of the function f_s.
$\vec{v}_t \in \mathbb{R}^d$	The vector representation of a token t.
$\vec{v'}_t \in \mathbb{R}^d$	Another vector of token t, used for prediction.

TABLE 5: Parameters to be estimated in training.

All $\vec{\theta}_{f_s}$ and $\vec{v}_t$ are initialized to small random value around zero. All $\vec{v'}_t$ are initialized to zeros. We use $2 \times d$ for f_s since we concatenate the vector for operation and operands to represent an instruction. We also define the following symbols according to the syntax of assembly language:

$\mathcal{S}(f_s) = seq[1:i]$	Multiple sequences generated from f_s.
$\mathcal{I}(seq_i) = in[1:j]$	Instructions of a sequence seq_i.
in_j	The j^{th} instruction in a sequence.
$\mathcal{A}(in_j)$	Operands of instruction in_j.
$\mathcal{P}(in_j)$	The operation of instruction in_j.
$\mathcal{T}(in_j)$	Represent the tokens of in_j.
$\mathcal{CT}(in_j) \in \mathbb{R}^{2\times d}$	Vector representation of an instruction in_j.
$\mathcal{CT}(in_{j-1}) \in \mathbb{R}^{2\times d}$	Vector representation of in_j's previous instruction.
$\mathcal{CT}(in_{j+1}) \in \mathbb{R}^{2\times d}$	Vector representation of an instruction in_j's next instruction.
$\delta(in_j, f_s)$	Vector representation of the joint memory of function f_s and in_j's neighbor instructions.

TABLE 6: Intermediate symbols used in training.

For an instruction in_j, We treat the concatenation of its operation and operands as its tokens $\mathcal{T}(in_j)$: $\mathcal{T}(in_j) = \mathcal{P}(in_j) \mid\mid \mathcal{A}(in_j)$, where $\mid\mid$ denotes concatenation. $\mathcal{CT}(in)$ denotes the vector representation of an instruction in.

$$\mathcal{CT}(in) = \vec{v}_{\mathcal{P}(in)} \mid\mid \frac{1}{|\mathcal{A}(in)|} \sum_{t}^{\mathcal{A}(in)} \vec{v}_{t_b}$$

The representation is calculated by averaging the vector representations of its operands $\mathcal{A}(in)$. The averaged vector is then concatenated to the vector representation $\vec{v}_{\mathcal{P}(in)}$ of its operation $\mathcal{P}(in)$.

As presented in Algorithm 1, the training procedure goes through each assembly function f_s in the repository and generates multiple sequences by calling $\mathcal{S}(f_s)$. For each sequence seq_i of function f_s, the neural network walks through the instructions from its beginning. We collect the current instruction in_j, its previous instruction in_{j-1}, and its next instruction in_{j+1}. We ignore the instructions that are out-of-boundary. We calculate $\mathcal{T}(in_{j-1})$ and $\mathcal{T}(in_{j+11})$ using the previous equation. By averaging f_s's vector representation $\vec{\theta}_{f_s}$ with $\mathcal{CT}(in_j - 1)$ and $\mathcal{CT}(in_j + 1)$, $\delta(in, f_s)$ models the joint memory of neighbor instructions:

$$\delta(in_j, f_s) = \frac{1}{3}(\vec{\theta}_{f_s} + \mathcal{CT}(in_{j-1}) + \mathcal{CT}(in_{j+1}))$$

For the current instruction in_j, the proposed model maximizes the following log probability:

$$\arg\max \sum_{t_c}^{\mathcal{T}(in_j)} log\ \mathbf{P}(t_c|f_s, in_{j-1}, in_{j+1})$$

It predicts each token in the current instruction in_j based on the joint memory of its corresponding function vector and its neighbor instruction vectors, as illustrated in Figure 5.

To model the above prediction, one can use a typical softmax multi-class classification layer and maximize the following log probability:

$$\begin{aligned}\mathbf{P}(t_c|\delta(in_j, f_s)) &= \mathbf{P}(\vec{v'}_{t_c}|\delta(in_j, f_s)) \\ &= \frac{f(\vec{v'}_{t_c}, \delta(in_j, f_s))}{\sum_d^D f(\vec{v'}_{t_d}, \delta(in_j, f_s))} \\ f(\vec{v'}_{t_c}, \delta(in_j, f_s)) &= Uh((\vec{v'}_{t_c})^T \times \delta(in_j, f_s))\end{aligned}$$

D denotes the whole vocabulary constructed upon the repository RP. $Uh(\cdot)$ denotes a sigmoid function applied to each value of a vector. The total number of parameters to be estimated is $(|D| + 1) \times 2 \times d$ for each pass of the softmax layout. The term $|D|$ is too large to be efficient for the softmax classification.

Therefore we use the k negative sampling approach [20], [21], to approximate the log probability:

$$\begin{aligned}&log\ \mathbf{P}(t_c|\delta(in_j, f_s)) \approx log\ f(\vec{v'}_{t_c}|\delta(in_j, f_s)) \\ &+ \sum_{i=1}^{k} \mathbb{E}_{t_d \backsim P_n(t_c)} \big([\![t_d \neq t_c]\!] log\ f(-1 \times \vec{v'}_{t_d},\ \delta(in_j, f_s)) \big)\end{aligned}$$

By manipulating the value of the parameters listed in Table 5, we can maximize the sum of the above log-probability for all the instruction in_j.

We follow the parallel stochastic gradient decent algorithm. In a single training step, we only consider a single token t_c of the current instruction in_j. We calculate the above log probability and its gradients with respect to the parameters that we are trying to manipulate. The gradients define the direction in which we should manipulate the parameters in order to maximize the log probability. The gradients are calculated by taking the derivatives with respect to each parameter defined in Table 5. The table below defines the symbol of the gradients:

$\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta)$	The gradient for current function f_s's $\vec{\theta}_{f_s}$.
$\frac{\partial}{\partial \vec{v'}_t} J(\theta)$	The gradient for the token t_c of current instruction in_j.
$\frac{\partial}{\partial \vec{v}_{\mathcal{P}(in_{j+1})}}$	The gradient for the operation of instruction in_{j+1}.
$\frac{\partial}{\partial \vec{v}_{\mathcal{P}(in_{j-1})}}$	The gradient for the operation of instruction in_{j-1}.
$\frac{\partial}{\partial \vec{v}_{t_b}} J(\theta)$	The gradient for each operation of instruction in_{j+1} and in_{j-1}.

TABLE 7: Gradients to be calculated in a training step.

The equations below calculate the gradients defined above.

	GCC O0	GCC O1	GCC O2	GCC O3
BusyBox	52,118	46,519	47,272	62,069
CoreUtils	38,176	36,168	35,117	41,421
Libgmp	12,919	15,534	14,602	16,234
ImageMagick	85,191	88,342	84,395	93,421
Libcurl	17,969	14,097	13,483	15,371
LibTomCrypt	12,021	10,135	10,258	13,451
OpenSSL	52,063	44,527	44,642	50,043
SQLite	27,621	24,978	29,332	38,699
zlib	2,898	2,747	2,668	3,706
PuTTYgen	5,495	4,957	5,065	7, 231
Total	306,471	288,004	286,834	341,646

TABLE 8: Number of basic blocks for each selected library compiled using different optimization options.

	Original	BCF	FLA	SUB	All
Libgmp	20,168	54,738	103,258	20,168	55,007
ImageMagick	83,704	218,315	434,599	83,702	216,904
LibTomCrypt	10,044	19,534	35,608	10,115	62,895
OpenSSL	46,298	100,315	160,265	46,278	289,657
Total	160,214	392,902	733,730	160,263	624,463

TABLE 9: Number of basic blocks for each selected library under different code obfuscation options.

$$\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta) = \frac{1}{3} \sum_{i}^{k} \mathbb{E}_{t_b \backsim P_n(t_c)} \big([\![t_b = t_c]\!] - f(\vec{v'}_t, \delta(in_j, f_s)) \big) \times \vec{v'}_t$$

$$\frac{\partial}{\partial \vec{v'}_t} J(\theta) = [\![t = t_c]\!] - f(\vec{v'}_t, \delta(in_j, f_s)) \times \delta(in_j, f_s)$$

It will be the same equation for the previous instruction in_{j-1}, by replacing in_{j+1} with in_{j-1}.

$$\frac{\partial}{\partial \vec{v}_{\mathcal{P}(in_{j+1})}} J(\theta) = \big(\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta) \big) [0 : d-1]$$

$$\frac{\partial}{\partial \vec{v}_{t_b}} J(\theta) = \frac{1}{|\mathcal{A}(in_{j+1})|} \times \big(\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta) \big) [d : 2d-1]$$

$$t_b \in \mathcal{A}(in_{j+1})$$

After, we use back propagation to update the values of all the involved parameters according to their gradients in Table 7, with a learning rate.

Appendix B. Extended Descriptive Statistics of the Dataset

This appendix provides additional descriptive statistics on the experimental dataset used in Section 5.1, Section 5.2, and Section 5.3

In the compiler optimization experiment (Section 5.1, *ImageMagick* generally has the largest number of assembly basic blocks while *zlib* has the least. By adopting different compiler optimization options, the generated number of basic blocks greatly varies. Specifically, O0 is very different from the other optimization levels. O1 and O2 appear to share a similar number. O3 has the largest number of basic blocks, which is generated by intensive inlining. Figure 12 shows the empirical distribution of the assembly functions length under different optimization levels. O3 tends to produce assembly functions that are much longer than O0, O1, and O2. O1 and O2 share similar distributions on function length.

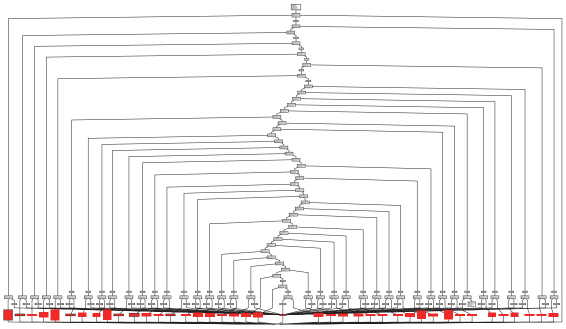
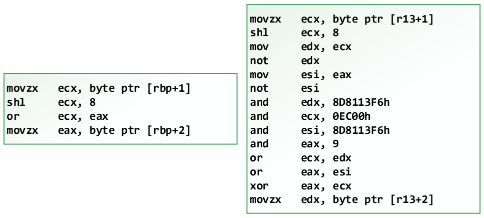


Figure 10: A function obfuscated by *O-LLVM* Control Flow Graph Flattening. Only the penultimate level (red filled) basic blocks contains the modified original logics.

In the *O-LLVM* obfuscation experiment (Section 5.2), we evaluate the the clone search methods before and after obfuscation. *O-LLVM* significantly increases the complexity of the binary code. Table 9 shows how the number of basic blocks have been changed across different obfuscation level. Figure 13 shows the empirical distribution of the assembly functions length under different obfuscation options. There are three different techniques and their combination:

- *BCF* modifies the control flow graph by adding a large number of irrelevant random basic blocks and branches. It will also split, merge, and reorder the original basic blocks. It almost double the number of basic blocks after obfuscation (see Table 9).
- *FLA* reorganizes the original CFG using complex hierarchy of new conditions as switches (see an example in Figure 1). Only the penultimate level of the CFG contains the modified original logics. It completely destroys the original CFG graph. The obfuscated binary on average contains 4 times of basic blocks than the original.



```
movzx   ecx, byte ptr [rbp+1]
shl     ecx, 8
or      ecx, eax
movzx   eax, byte ptr [rbp+2]
```

```
movzx   ecx, byte ptr [r13+1]
shl     ecx, 8
mov     edx, ecx
not     edx
mov     esi, eax
not     esi
and     edx, 8D8113F6h
and     ecx, 0EC00h
and     esi, 8D8113F6h
and     eax, 9
or      ecx, edx
or      eax, esi
xor     eax, ecx
movzx   edx, byte ptr [r13+2]
```

Figure 11: An assembly fragment obfuscated by *O-LLVM* Instruction Substitution. Left: the original fragment. Right: the obfuscated fragment.

- *SUB* substitutes fragments of assembly code to its equivalent form by going one pass over the function logic using predefined rules. This technique modifies the contents of basic blocks and adds new constants. *SUB* does not change much of the graph structure Figure 11 shows an example. Figure 13 shows that it increase the length of the original assembly function.

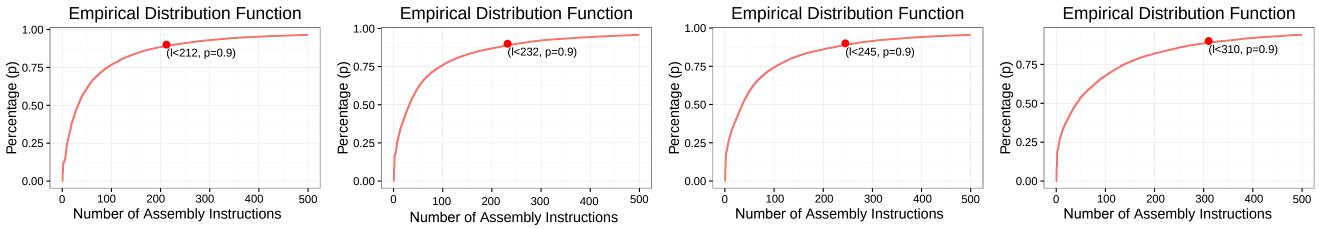


Figure 12: The empirical distribution function of the assembly function length in terms of number of instructions. From left to right, each diagram corresponds to the optimization options O0, O1, O2, and O3.

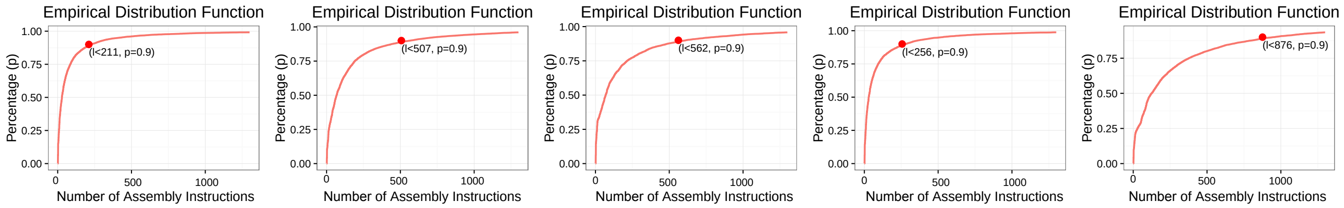


Figure 13: The empirical distribution function of the assembly function length in terms of the number of instructions. Each diagram from left to right corresponds to the original binary, the obfuscation techniques Bogus Control Flow Graph, Control Flow Flattening, Instruction Substitution, and the application of all the obfuscation techniques provided by *Obfuscator-LLVM*.