Kam1n0: MapReduce-based Assembly Clone Search for Reverse Engineering

Steven H. H. Ding* Benjamin C. M. Fung* Philippe Charland†

*School of Information Studies, McGill University, Montreal, QC, Canada
†Mission Critical Cyber Security Section, Defence R&D Canada - Valcartier, Quebec, QC, Canada
steven.h.ding@mail.mcgill.ca ben.fung@mcgill.ca philippe.charland@drdc-rddc.gc.ca

ABSTRACT

Assembly code analysis is one of the critical processes for detecting and proving software plagiarism and software patent infringements when the source code is unavailable. It is also a common practice to discover exploits and vulnerabilities in existing software. However, it is a manually intensive and time-consuming process even for experienced reverse engineers. An effective and efficient assembly code clone search engine can greatly reduce the effort of this process, since it can identify the cloned parts that have been previously analyzed. The assembly code clone search problem belongs to the field of software engineering. However, it strongly depends on practical nearest neighbor search techniques in data mining and databases. By closely collaborating with reverse engineers and *Defence Research and Development Canada* (*DRDC*), we study the concerns and challenges that make existing assembly code clone approaches not practically applicable from the perspective of data mining. We propose a new variant of LSH scheme and incorporate it with graph matching to address these challenges. We implement an integrated assembly clone search engine called *Kam1n0*. It is the first clone search engine that can efficiently identify the given query assembly function's *subgraph clones* from a large assembly code repository. Kam1n0 is built upon the Apache Spark computation framework and Cassandra-like key-value distributed storage. A deployed demo system is publicly available.[1] Extensive experimental results suggest that Kam1n0 is accurate, efficient, and scalable for handling large volume of assembly code.

Keywords

Assembly clone search; Information retrieval; Mining software repositories

[1]Kam1n0 online demo (no installation required). Both the user name and password are "sigkdd2016". Use Chrome for best experience. http://dmas.lab.mcgill.ca/projects/kam1n0.htm

KDD '16 August 13–17, 2016, San Francisco, CA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4232-2/16/08...$15.00

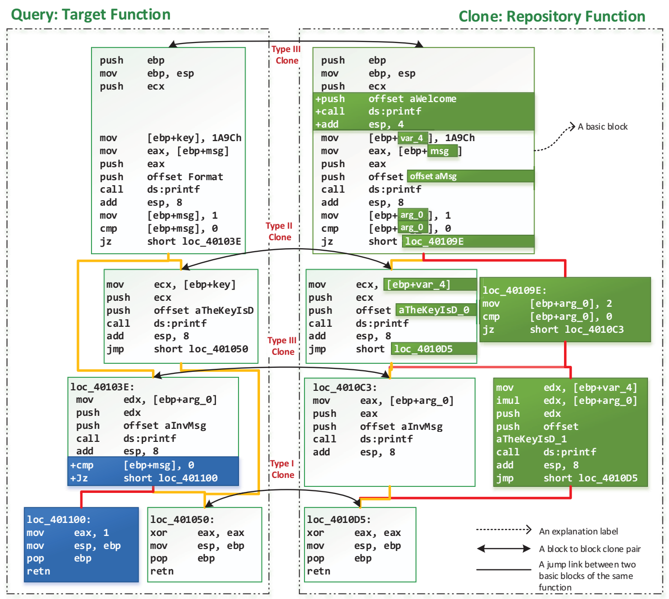
http://dx.doi.org/10.1145/2939672.2939719

1. INTRODUCTION

Code reuse is a common but uncontrolled issue in software engineering [15]. Mockus [25] found that more than 50% of files were reused in more than one open source project. Sojer's survey [29] indicates that more than 50% of the developers modify the components before reusing them. This massively uncontrolled reuse of source code does not only introduce legal issues such as GNU General Public License (GPL) violation [36, 17]. It also implies security concerns, as the source code and the vulnerabilities are uncontrollably shared between projects [4].

Identifying all these infringements and vulnerabilities requires intensive effort from reverse engineers. However, the learning curve to master reverse engineering is much steeper than for programming [4]. Reverse engineering is a time consuming process which involves inspecting the execution flow of the program in assembly code and determining the functionalities of the components. Given the fact that code reuse is prevalent in software development, there is a pressing need to develop an efficient and effective assembly clone search engine for reverse engineers. Previous clone search approaches only focus on the search accuracy. However, designing a practically useful clone search engine is a nontrivial task which involves multiple factors to be considered. By closely collaborating with reverse engineers and *Defence Research and Development Canada* (*DRDC*), we outline the deployment challenges and requirements as follows:

Interpretability and usability: An assembly function can be represented as a control flow graph consisting of connected basic blocks. Given an assembly function as query, all of the previous assembly code clone search approaches [7, 6, 18, 26] only provide the top-listed candidate assembly functions. They are useful when there exists a function in the repository that shares a high degree of similarity with the query. However, due to the unpredictable effects of different compilers, compiler optimization, and obfuscation techniques, given an unknown function, it is less probable to have a very similar function in the repository. Returning a list of clones with a low degree of similarity values is not useful. As per our discussions with DRDC, a practical search engine should be able to decompose the given query assembly function to different known subgraph clones which can help reverse engineers better understand the function's composition. We define a *subgraph clone* as one of its subgraphs that can be found in the other function. Refer to the example in Figure 1. The previous clone search approaches cannot address this challenge.


Query: Target Function
Clone: Repository Function
Type III Clone
Type II Clone
Type III Clone
Type I Clone
A basic block
An explanation label
A block to block clone pair
A jump link between two basic blocks of the same function

```
push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+key], 1A9Ch
mov     eax, [ebp+msg]
push    eax
push    offset Format
call    ds:printf
add     esp, 8
mov     [ebp+msg], 1
cmp     [ebp+msg], 0
jz      short loc_40103E

mov     ecx, [ebp+key]
push    ecx
push    offset aTheKeyIsD
call    ds:printf
add     esp, 8
jmp     short loc_401050

loc_40103E:
 mov    edx, [ebp+arg_0]
 push   edx
 push   offset aInvMsg
 call   ds:printf
 add    esp, 8
+cmp    [ebp+msg], 0
+Jz     short loc_401100

loc_401100:
mov     eax, 1
mov     esp, ebp
pop     ebp
retn

loc_401050:
xor     eax, eax
mov     esp, ebp
pop     ebp
retn
```

```
 push   ebp
 mov    ebp, esp
 push   ecx
+push   offset aWelcome
+call   ds:printf
+add    esp, 4
 mov    [ebp+var_4], 1A9Ch
 mov    eax, [ebp+msg]
 push   eax
 push   offset offset aMsg
 call   ds:printf
 add    esp, 8
 mov    [ebp+arg_0], 1
 cmp    [ebp+arg_0], 0
 jz     short loc_40109E

mov     ecx, [ebp+var_4]
push    ecx
push    offset aTheKeyIsD_0
call    ds:printf
add     esp, 8
jmp     short loc_4010D5

loc_40109E:
mov     [ebp+arg_0], 2
cmp     [ebp+arg_0], 0
jz      short loc_4010C3

loc_4010C3:
mov     eax, [ebp+arg_0]
push    eax
push    offset aInvMsg
call    ds:printf
add     esp, 8

mov     edx, [ebp+var_4]
imul    edx, [ebp+arg_0]
push    edx
push    offset
aTheKeyIsD_1
call    ds:printf
add     esp, 8
jmp     short loc_4010D5

loc_4010D5:
xor     eax, eax
mov     esp, ebp
pop     ebp
retn
```

Figure 1: An example of the clone search problem. Basic blocks with a white background form a *subgraph clone* between two functions. Three types of code clones are considered in this paper. Type I: literally identical; Type II: syntactically equivalent; and Type III: minor modifications.

Efficiency and Scalability: An efficient engine can help reverse engineers retrieve clone search results on-the-fly when they are conducting an analysis. Instant feedback tells the reverse engineer the composition of a given function that is under investigation. Scalability is a critical factor as the number of assembly functions in the repository needs to scale up to millions. The degradation of search performance against the repository size has to be considered. Previous approaches in [6, 7], which trade efficiency and scalability for better accuracy, have a high latency for queries and are thus not practically applicable.

Incremental updates: The clone search engine should support incremental updates to the repository, without re-indexing existing assembly functions. [7] requires median statistics to index each vector and [26] requires data-dependent settings for its index, so they do not satisfy this requirement.

Clone search quality: Practically, clones among assembly functions are one-to-many mappings, i.e., a function has multiple cloned functions with different degrees of similarity. However, all previous approaches [6, 7, 18, 26] assume that clones are one-to-one mappings in the experiment. This is due to the difficulty of acquiring such a one-to-many labeled dataset. Moreover, they use different evaluation metrics. Therefore, it is difficult to have a direct comparison among them with respect to the search quality. We need to develop a one-to-many labeled dataset and an unified evaluation framework to quantify the clone search quality.

To address the above requirements and challenges, we propose a new variant of LSH scheme and incorporate it with a graph matching technique. We also develop and deploy a new assembly clone search engine called *Kam1n0*. Our major contributions can be summarized as follows:

- **Solved a challenging problem for the reverse engineering community:** Kam1n0 is the first assembly code clone search engine that supports subgraph clone search. Refer to the example in Figure 1. It promotes the interpretability and usability by providing subgraph clones as results, which helps reverse engineers analyzing new and unknown assembly functions. Kam1n0 won the second prize at the 2015 Hex-Rays plugin Contest[2] and its code is publicly accessible on GitHub.[3]

- **Efficient inexact assembly code search:** The assembly code vector space is highly skewed. Small blocks tend to be similar to each other and large blocks tend to be sparsely distributed in the space. Original hyperplane hashing with banding technique equally partitions the space and does not handle the unevenly distributed data well. We propose a new *adaptive locality sensitive hashing* (*ALSH*) scheme to approximate the cosine similarity. To our best knowledge, ALSH is the first incremental locality sensitive hashing scheme that solves this issue specifically for cosine space with theoretical guarantee. It retrieves fewer points for dense areas and more points for sparse ones in the cosine space. It is therefore efficient in searching nearest neighbors.

- **Scalable sub-linear subgraph search:** We propose a *MapReduce* subgraph search algorithm based on the *Apache Spark* computational framework without an additional index. Different to the existing subgraph isomorphism search problem in data mining, we need to retrieve subgraphs that are both isomorphic to the query and the repository functions as graphs. Thus, existing algorithms are not directly applicable. Algorithmically, our approach is bounded by polynomial complexity. However, our experiment suggests that it is sub-linear in practice.

- **Accurate and robust function clone search:** Kam1n0 is the first approach that integrates both the inexact assembly code and subgraph search. Previous solutions do not consider both of them together. Our extensive experiments suggest that it boosts the clone search quality and yields stable results across different datasets and metrics.

- **Develop a labeled dataset and benchmark state-of-the-art assembly code clone solutions.** We carefully construct a new *labeled* one-to-many assembly code clone dataset that is available to the research community by linking the source code and assembly function level clones. We benchmark and report the performance of twelve existing state-of-the-art solutions with Kam1n0 on the dataset using several metrics. We also setup a mini-cluster to evaluate the scalability of Kam1n0.

The remainder of this paper is organized as follows. Section 2 situates our study within the literature of three different research problems. Section 3 formally defines the studied problem. Section 4 provides an overview of our solution and system design. Section 5 presents the preprocessing steps and the chosen vector space. Section 6 introduces our proposed locality sensitive hashing scheme. Section 7 presents our graph search algorithm. Section 8 presents our benchmark experiments. Section 9 provides the conclusion.

[2]https://hex-rays.com/contests/2015/#kam1n0
[3]https://github.com/McGill-DMaS/Kam1n0-Plugin-IDA-Pro

2. RELATED WORK

Locality sensitive hashing. *Locality sensitive hashing* (*LSH*) has been studied for decades to solve the ϵ-approximated Nearest Neighbor (ϵNN) problem, since exact nearest neighbor is not scalable to high dimensional data. One of the prevalent problems of LSH is the *uneven data distribution issue*, as LSH equally partitions the data space. To mitigate this issue, several approaches have been proposed including the *LSH-Forest* [2], *LSB-Forest* [32], *C2LSH* [10] and *SK-LSH* [22]. It has been shown that the cosine vector space is robust to different compiler settings [18] in assembly code clone search. However, LSH-Forest, C2LSH and SK-LSH are designed for the p-stable distribution, which does not fit the cosine space. LSB-Forest dynamically and unequally partition the data space. As pointed out by [33], it requires the hash family to possess the $(\epsilon, f(\epsilon))$ property. However, to our best knowledge, such a family in cosine space is still unknown. There are other learning-based approaches [11], which do not meet our incremental requirement. Wang et al. [35] provide a more comprehensive survey on LSH. To satisfy our requirements, we propose the ALSH scheme specifically for the cosine space. Different to the LSH-Forest, ALSH takes more than one bit when going down the tree structure and does not require the $(\epsilon, f(\epsilon))$ property for the LSH family to have theoretical guarantee. Unlike LSB forest [2], we dynamically construct the buckets to adapt to different data distributions.

Subgraph isomorphism. Ullmann [34] proposed the first practical subgraph isomorphism algorithm for small graphs. Several approaches were proposed afterwards for large scale graph data, such as *TurboISO* [13] and *STwig* [31]. It has been shown that they can solve the subgraph isomorphism problem in a reasonable time. However, they do not completely meet our problem settings. The subgraph isomorphism problem needs to retrieve subgraphs that are isomorphic to the graphs in the repository and identical to the query. However, we need to retrieve subgraphs that are isomorphic to the graphs in the repository and isomorphic to the graph of the query, which significantly increases the complexity. More details will be discussed in Section 7. Such a difference requires us to propose a specialized search algorithm. Lee et al. [21] provide a comprehensive survey and performance benchmark on subgraph isomorphism.

Assembly code clone search. The studies on the assembly code clone search problem are recent. Only a few approaches exist [6, 7, 18, 26]. They all rely on the inexact text search techniques of data mining. *BinClone* [7] models assembly code into an Euclidean space based on frequency values of selected features. It is inefficient and not scalable due to the exponential 2-combination of features which approximates the 2-norm distance. *LSH-S* in [26] models assembly code into a cosine space based on token frequency and approximate the distance by hyperplane hashing and banding scheme. It equally partitions the space and suffers from the uneven data distribution problem. *Graphlet* [18] models assembly code into a cosine space based on extracted signatures from assembly code. However, it cannot detect any subgraph clones that is smaller than the graphlet size. *Tracelet* [6] models assembly code according to string editing distance. It compares function one by one, which is not scalable. Kam1n0 is fundamentally different to the previous approaches. It is an integration of inexact assembly code search and the subgraph search. It enables clone subgraph search of any size.

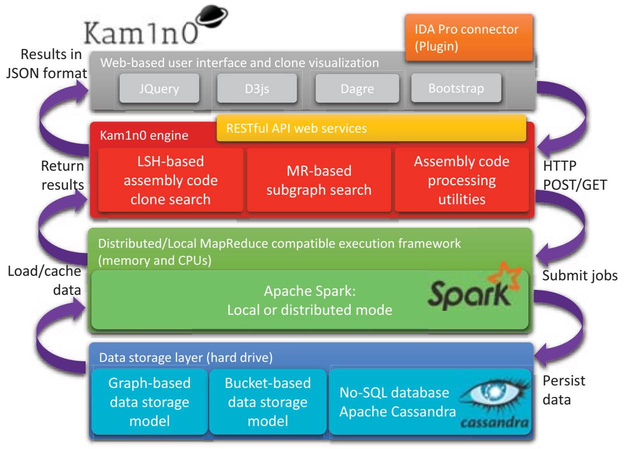

Kam1n0
IDA Pro connector (Plugin)
Results in JSON format
Web-based user interface and clone visualization
JQuery
D3js
Dagre
Bootstrap
Kam1n0 engine
RESTful API web services
Return results
LSH-based assembly code clone search
MR-based subgraph search
Assembly code processing utilities
HTTP POST/GET
Distributed/Local MapReduce compatible execution framework (memory and CPUs)
Load/cache data
Apache Spark: Local or distributed mode
Spark
Submit jobs
Data storage layer (hard drive)
Graph-based data storage model
Bucket-based data storage model
No-SQL database Apache Cassandra
cassandra
Persist data

Figure 2: The overall solution stack of the Kam1n0 engine.

3. PROBLEM STATEMENT

Reverse engineering starts from a binary file. After being unpacked and disassembled, it becomes a list of assembly functions. In this paper, *function* represents an assembly function; *block* represents a basic block; *source function* represents the actual function written in source code, such as C++; *repository function* stands for the assembly function that is indexed inside the repository; *target function* denotes the assembly function that is given as a query; and correspondingly, *repository blocks* and *target blocks* refer to their respective basic blocks. Each function f is represented as a control flow graph denoted by (B, E), where B indicates its basic blocks and E, indicates the edges that connect the blocks. Let $B(\mathrm{RP})$ be the complete set of basic blocks in the repository and $F(\mathrm{RP})$ be the complete set of functions in the repository. Given an assembly function, our goal is to search all its subgraph clones inside the repository RP. We formally define the search problem as follows:

Definition 1. (*Assembly function subgraph clone search*) Given a target function f_t and its control flow graph (B_t, E_t), the search problem is to retrieve all the repository functions $f_s \in \mathrm{RP}$, which share at least one subgraph clone with f_t. The shared list of subgraph clones between f_s and f_t is denoted by $sg_s[1 : a]$, where $sg_s[a]$ represents one of them. A subgraph clone is a set of basic block clone pairs $sg_s[a] = \{\langle b_t, b_s \rangle, \dots\}$ between f_s and f_t, where $b_t \in B_t$, $b_s \in B_s$, and $\langle b_t, b_s \rangle$ is a type I, type II, or type III clone (see Figure 1). Formally, given f_t, the problem is to retrieve all $\{f_s | f_s \in \mathrm{RP} \text{ and } |sg_s| > 0\}$. □

4. OVERALL ARCHITECTURE

The Kam1n0 engine is designed for general key-value storage and the *Apache Spark*[4] computational framework. Its solution stack, as shown in Figure 2, consists of three layers. The data storage layer is concerned with how the data is stored and indexed. The distributed/local execution layer manages and executes the jobs submitted by the Kam1n0

[4]Apache Spark, available at: http://spark.apache.org/

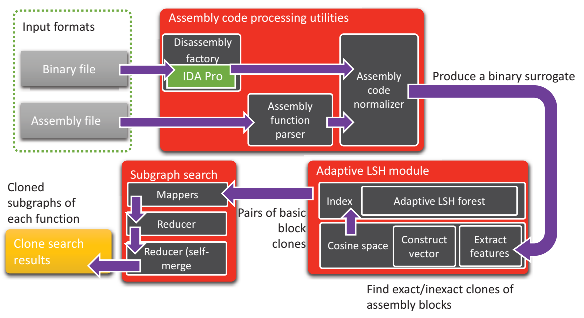


Figure 3: Assembly clone search data flow.

engine. The Kam1n0 engine splits a search query into multiple jobs and coordinates their execution flow. It also provides the RESTful APIs. We have implemented a web-based user interface and an *Hex-Rays IDA Pro* plugin[5] as clients. IDA Pro is a popular interactive disassembler that is used by reverse engineers.

Figure 3 depicts the data flow of the clone search process. It consists of the following steps. *Preprocessing*: After parsing the input (either a binary file or assembly functions) into control flow graphs, this step normalizes assembly code into a general form, which will be elaborated in Section 5. *Find basic blocks clone pairs:* Given a list of assembly basic blocks from the previous step, it finds all the clone pairs of blocks using ALSH. *Search the subgraph clones:* Given the list of clone block pairs, the MapReduce module merges and constructs the subgraph clones. Note that this clone search process does not require any source code.

5. PREPROCESSING AND VECTOR SPACE

We choose the cosine vector space to characterize the semantic similarity of assembly code. It has been shown that the cosine vector space is robust to different compiler settings [18]. It can mitigate the linear transformation of assembly code. For example, to optimize the program for speed, the compiler may unroll and flatten a loop structure in assembly code by repeating the code inside the loop multiple times. In this case, the cosine similarity between the unrolled and original loop is still high, due to the fact that the cosine distance only considers the included angle between two vectors. The features selected in Kam1n0 include mnemonics, combinations of mnemonics and operands, as well as mnemonics n-gram, which are typically used in assembly code analysis [7, 26]. The equivalent assembly code fragments can be represented in different forms. To mitigate this issue, we normalize the operands in assembly code during the preprocessing. We extend the normalization tree used in BinClone [7] with more types. There are three normalization levels: *root*, *type*, and *specific*. Each of them corresponds to a different generalization level of assembly code. More details can be found in our technical report.[6]

6. LOCALITY SENSITIVE HASHING

In this section, we introduce an *Adaptive Locality Sensitive Hashing* (*ALSH*) scheme for searching the block-level semantic clones. As discussed in Section 2, exact nearest neighbor search is not scalable. Thus, we started from the reduction of the ϵ-approximated k-NN problem:

[5]IDA Pro, available at: http://www.hex-rays.com/
[6]Kam1n0 technical report, available at: http://dmas.lab.mcgill.ca/fung/pub/DFC16kdd.pdf

Definition 2. (ϵ*-approximated NN search problem*) Given a dataset $D \subset \mathbb{R}^d$ ($\mathbb{R}$ denotes real numbers) and a query point q, let r denote the distance between the query point q and its nearest neighbor o^*. This problem is to find an approximated data point within the distance $\epsilon \times r$ where $\epsilon > 1$ □

The ϵ-approximated k-NN search problem can be reduced to the ϵNN problem by finding the k data points where each is an ϵ-approximated point of the exact k-NN of q [10]. The locality sensitive hashing approaches do not solve the ϵNN problem directly. ϵNN is further reduced into another problem: (ϵ, r)-approximated ball cover problem [1, 14].

Definition 3. (*(ϵ, r)-approximated ball cover problem*) Given a dataset $D \subset \mathbb{R}^d$ and a query point q, let $B(q, r)$ denote a ball with center q and radius r. The query q returns the results as follows:

- if there exists a point $o^* \in B(q, R)$, then return a data point from $B(q, \epsilon R)$
- if $B(q, \epsilon R)$ does not contain any data object in D, then return nothing. □

One can solve the (ϵ, r) ball cover problem by using the Locality Sensitive Hashing (LSH) families. A locality sensitive hashing family consists of hashing functions that can preserve the distance between points.

Definition 4. (*Locality Sensitive Hashing Family*) Given a distance r under a specific metric space, an approximation ratio ϵ, and two probabilities $p_1 > p_2$, a hash function family $\mathcal{H} \to \{h : \mathbb{R}^d \to U\}$ is $(r, \epsilon r, p1, p2)$-sensitive such that:

- if $o \in B(q, r)$, then $Pr_{\mathcal{H}}[h(q) = h(o)] \geqslant p_1$
- if $o \notin B(q, \epsilon r)$, then $Pr_{\mathcal{H}}[h(q) = h(o)] \leqslant p_2$ □

LSH families are available for many metric spaces such as cosine similarity [3], hamming distance [14], Jaccard coefficient, and p-stable distributions [5]. Based on our chosen cosine vector space, we adopt the random hyperplane hash [3] family, where *sign*$(\cdot)$ output the sign of the input.

$$h(\vec{o}) = sign(\vec{o} \cdot \vec{a}) \tag{1}$$

By substituting the random vector $\vec{a}$ we can obtain different hash functions in the family. The collision probability of two data points $\vec{o_1}$ and $\vec{o_2}$ on Equation 1 can be formulated as:

$$P[h(\vec{o_1}) = h(\vec{o_2})] = 1 - \frac{\theta_{\vec{o_1},\vec{o_2}}}{\pi} \tag{2}$$

$\theta_{\vec{o_1},\vec{o_2}}$ is the included angle between $\vec{o_1}$ and $\vec{o_2}$. The probability that two vectors have the same projected direction on a random hyperplane is high when their included angle is small.

THEOREM 1. *The random hyperplane hash function is a* $(r, \epsilon r, 1 - r/\pi, 1 - \epsilon r/\pi)$ *sensitive hashing family.* □

PROOF. According to Definition 4 and Equation 2: $p_1 = 1 - r/\pi$ and $p_2 = 1 - \epsilon r/\pi$ □

To use the locality sensitive hashing families to solve the ball cover problem, it needs a hashing scheme to meet the quality requirement. The *E2LSH* approach was originally proposed by [14] and extended by[5]. It concatenates k different hash functions $[h_1, \ldots, h_k]$ from a given LSH family $\mathcal{H}$ into a function $g(o) = (h_1(o), \ldots, h_k(o))$, and adopts l such functions. The parameters k and l are chosen to ensure the following two properties are satisfied:

Property 1. (P_1): if there exists $p^* \in B(q, r)$, then $g_j(p^*) = g_j(q)$ from some $j = 1 \ldots l$. □

Property 2. (P_2): the total number of points $\notin B(q, \epsilon r)$ that collides with q is less than $2l$. □

It is proven that if the above two properties hold with constant probability, the algorithm can correctly solve the (ϵ, r)-approximated ball cover problem [14]. For E2LSH, by picking $k = log_{p_2}(1/n)$ and $l = n^{\rho}$ where $\rho = \frac{ln 1/p_1}{ln 1/p_2}$, both Properties 1 and 2 hold with constant probability.

However, the ball cover problem is a strong reduction to the NN problem since it adopts the same radius r for all points. Real-life data cannot always be evenly distributed. Therefore, it is difficult to pick an appropriate r. We denote this as the *uneven data distribution issue*. In [12], a magic r_m is adopted heuristically. But as pointed out by [32], such a magic radius may not exist. A weaker reduction was proposed in [14], where the NN problem is reduced to multiple (r, ϵ)-NN ball cover problems with varying $r = \{1, \epsilon^2, \epsilon^3, \dots\}$. The intuition is that points in different density areas can find a suitable r. However, such a reduction requires a large space consumption and longer response time. Other indexing structures have been proposed to solve this issue. Per our discussion in Section 2, existing techniques do not meet our requirement. Thus, we customize the LSH-forest approach and propose the ALSH structure.

6.1 Adaptive LSH Structure

We found that the limitation of the expanding sequence of r in previous section is too strong. It is unnecessary to *exactly* follow the sequence $r = \{1, \epsilon^2, \epsilon^3, \dots\}$, as long as r is increasing in a similar manner to $r_{t+1} = r_t \times \epsilon$. Thus, we customize the ϵ-approximated NN problem as follows:

Definition 5. (*$f(r)$-approximated NN search problem*) Given a dataset $D \subset \mathbb{R}^d$ and a query point q, let r denote the distance between the query point q and its nearest neighbor o^*. The problem is to find an approximated data point within the distance $f(r)$, where $f(r)/r > 1$ □

Instead of using a fix approximation ratio, we approximate the search by using a function on r. We issue a different sequence of expanding r. The expanding sequence of r is formulated as $r_0, r_1, \dots, r_t, r_{t+1}, \dots, r_m$, where $r_t < r_{t+1}$. Similar to the E2LSH approach, we concatenate multiple hash functions from the random hyperplane hash family $\mathcal{H}$ into one. However, we concatenate different number of hash functions for different values of r. This number is denoted by k_t for r_t, and the sequence of k is denoted by $k_0, k_1, \dots, k_t, k_{t+1}, \dots, k_m$, where $k_t > k_{t+1}$. Recall that the concatenated function is denoted by g. Consequently, there will be a different function g at position t, which is denoted by g_t. Yet, function g_t and function $g_{(t+1)}$ can share k_{t+1} hash functions. With p_m to be specified later, we set the r value at position t as follows:

$$r_t = \pi \times (1 - p_m^{(1/k_t)}) \tag{3}$$

This allows us to have the effect of increasing the r value by decreasing the k value. We calculate the value of k at position t as follows:

$$k_t = c \times k_{t+1}, \text{where} \quad c > 1 \tag{4}$$

By getting t_{t+1} from Equation 3, substituting k_t using Equation 4, and substituting p_m using Equation 3, we have:

$$r_{t+1} = \pi \times \left(1 - (1 - \frac{r_t}{\pi})^c\right) = f_c(r_t) \quad f_c(r_t)/r_t > 1 \tag{5}$$

By setting c equals to 2, we can get an approximately similar curve of r sequence to the original sequence $r_{t+1} = r_t \times \epsilon$

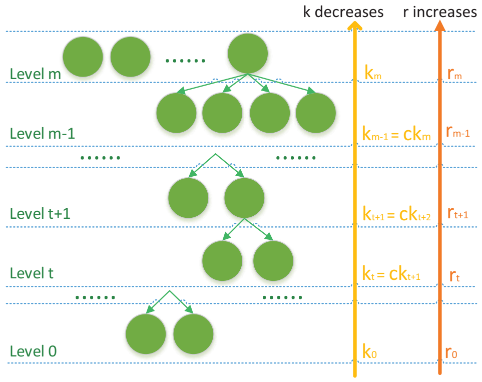


Figure 4: The index structure for the Adaptive Locality Sensitive Hashing (ALSH). There are $m+1$ levels on this tree. Moving from level t to level $t+1$ is equivalent of increasing the search radius from r_t to r_{t+1}.

where ϵ equals to 2. Following the aforementioned logic, we construct an Adaptive Locality Sensitive Hashing (ALSH) index in the form of prefix trees.

As shown in Figure 4, the index structure is a prefix tree of the signature values calculated by $\mathcal{G} = \{g_m, g_{m-1}, \dots, g_0\}$. Level t corresponds to the position t in the r expanding sequence. By introducing different values of k_t, each level represents a different radius r_t. Each level denotes a different g_t function and the g_t function is a concatenation of k_t hash functions. Moving up from a node at level t to its parent at level $t+1$ indicates that it requires a shorter matched prefix. The nodes which have the same parent at level t share the same prefix that is generated by g_t.

To locate the leaf for a given data point $q \in \mathbb{R}^d$, ALSH dynamically constructs the hash functions by trying $g_t \in \mathcal{G}$ in sequence. The signature of g_t can be generated by padding additional hash values to g_{t+1} since $k_t = c \times k_{t+1}$. Following [14, 32], with l to be specified later, we adopt l such prefix trees as the ALSH's index. Given a query point q, we first locate the corresponding leaves in all prefix trees. With l to be specified later, we collect the first $2l$ points from all the leaf buckets. To index a point, we locate its corresponding leaf in each tree and insert it to the leaf bucket. Suppose a leaf is on level t_{t+1}. If the number of points in that leaf is more than $2l$, we split all the data points of that leaf into the next level t by using g_t. All the trees are dynamically constructed based on the incoming points to be indexed in sequence. Therefore, they can be incrementally maintained. Unlike the learning-based LSH [11], Kam1n0 does not require the whole repository to estimate the hash functions to build the index.

It can be easily proved that g_t is a $(r_t, r_{t+1}, p_m, p_m^c)$-sensitive hash family and g_t can correctly solve the $(r_{t+1}/r_t, r_t)$-approximated ball cover problem by setting $p_m^c = 1/n$ and $l = n^{1/c}$. The proof follows [14]. Details and implementation on key-value data store can be found in our technical report.[6] Another parameter r_m controls the starting k_m value at the root value. It indicates the maximum distance that two points can be considered as valid neighbors. For sparse points far away from each other, they are not considered as neighbors unless their distance is within r_m.

For a single ALSH tree, the depth in the worst case is k_0, and all the leaves are at level 0. In this case, the tree is equivalent to the E2LSH with $k = k_0$. Therefore, the

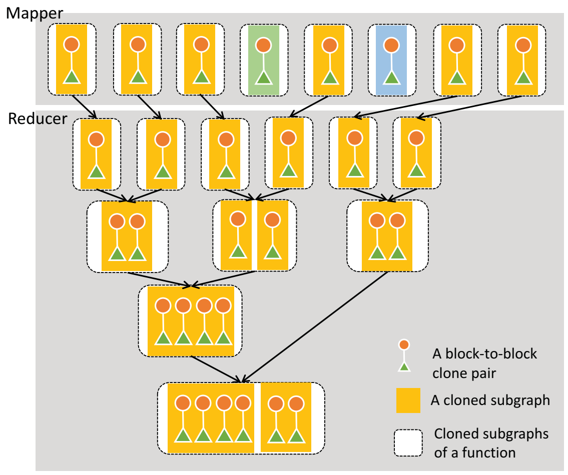


Figure 5: The MapReduce-based subgraph clone construction process.

space consumption for $l = n^{1/c}$ ALSH trees are bounded by $O(dn+n^{1+1/c})$, where $O(dn)$ is the data points in a dataset and $O(n^{1+1/c})$ is the space of indexes for the trees. The query time for a single ALSH prefix tree is bounded by its height. Given the maximum k value k_0 and the minimum k value k_m, its depth in the worst case is $log_c(k_0/k_m) + 1$. Thus, the query time for $l = n^{1/c}$ prefix trees is bounded by $O(log_c(k_0/k_m) \times n^{1/c})$. The ALSH index needs to build $n^{1/c}$ prefix trees for the full theoretical quality to be guaranteed. Based on our observation, setting l to 1 and 2 is already sufficient for providing good quality assembly code clones.

7. SUBGRAPH CLONE SEARCH

Even subgraph isomorphism is NP-hard in theory [21, 28], many algorithms have been proposed to solve it in a reasonable time. Formally, the subgraph isomorphism algorithm solved by most of these systems [13, 21, 31] is defined as:

Definition 6. (*Subgraph Isomorphism Search*) A graph is denoted by a triplet (V, E, L) where V represents the set of vertices, E represents the set of edges, and L represents the labels for each vertex. Given a query graph $q = (V, E, L)$ and a data graph $g = (V', E', L')$ a subgraph isomorphism (also known as *embedding*) is an injective function $M : V \to V'$ such that the following conditions hold: (1) $\forall u \in V, L(u) \in L'(M(u))$, (2) $\forall(u_i, u_j) \in E, (M(u_i), M(u_j)) \in E'$, and (3) $L(u_i, u_j) = L'(M(u_i), M(u_j))$. The search problem is to find all distinct embeddings of q in g. $\square$

The difference between this problem and ours in Definition 1 is two-fold. First, our problem is to retrieve *all the subgraph clones* of the target function f_t's control flow graph from the repository. In contrast, this problem only needs to retrieve the exact matches of query graph q within g. Refer to Conditions 1, 2, and 3 in Definition 6, or the termination condition of the procedure on Line 1 of subroutine *SubgraphSearch* in [21]. Our problem is more challenging and can be reduced to the problem in Definition 6 by issuing all the subgraphs of f_t as queries, which introduces a higher algorithmic complexity. Second, there is no such L data label attribute in our problem, but two types of edges: the control flow graph which links the basic blocks and the semantic relationship between basic blocks which is evaluated at the querying phase. Existing algorithms for subgraph isomorphism are not directly applicable. Assembly code control graphs are sparser than other graph data as there are less number of links between vertices and typically, basic blocks are only linked to each other within the same function. Given such properties, we can efficiently construct the subgraph clones respectively for each repository function f_s if it has more than one clone blocks in the previous step.

Algorithm 1 Mapper

Input A basic block clone pair $\langle b_t, b_s\rangle$
Output A pair consisting of $\langle f_s, sg_s\rangle$

```
1: $f_s \leftarrow$ getFunctionId($b_s$)
2: $sg_s \leftarrow [\,]$                        ▷ create an empty list of subgraph clones
3: $cloneGraph \leftarrow \{\langle b_t, b_s\rangle\}$   ▷ create a subgraph clone with one clone pair
4: $sg_s[0] \leftarrow cloneGraph$               ▷ list of subgraph clones but at this moment, it has only one.
5: return $\langle f_s, sg_s\rangle$ with $f_s$ as key index
```

7.1 MapReduce Subgraph Search

We adopt two functions in the Apache Spark MapReduce execution framework, namely the *map* function and the *reduce-by-key* function. In our case, the map function transforms the clone pairs generated by ALSH (refer to the data flow in Figure 3) and the reduce-by-key function constructs subgraph clone respectively for different repository function f_s. Figure 5 shows the overview of our subgraph clone search approach.

The signature for the map function (Algorithm 1) is $\langle b_t, b_s\rangle \to \langle f_s, sg_s[1 : a]\rangle$. Each execution of the map function takes a clone pair $\langle b_t, b_s\rangle$ produced by ALSH and transforms it to $\langle f_s, sg_s[1 : a]\rangle$, which is a pair of repository function id f_s and its list of subgraph clones sg_s in Definition 1. The map functions are independent to each other.

The outputs of the map functions correspond to the first row in Figure 5. A red circle represents a target basic block b_s and a green triangle represents a source basic block b_t. The link between them indicates that they are a block-to-block clone pair $\langle b_t, b_s\rangle$, which is produced in the previous step. A white rectangle represents a list of subgraph clones and the colored rectangle inside it represents a subgraph clone. Algorithm 1 maps each clone pair into a list of subgraph clones which contains only one subgraph clone. Each subgraph clone is initialized with only one clone pair.

After the map transformation functions, the reduce-by-key function reduces the produced lists of subgraph clones. The reducer merges a pair of lists sg_s^1 and sg_s^2 into a single one, by considering their subgraph clones' connectivity. The reduce process is executed for the links from the second row to the last row in Figure 5. Only the lists of subgraph clones with the same f_s will be merged. As indicated by the links between the first and second row in Figure 5, only rectangles with the same orange background can be reduced together. Rectangles with other background colors are reduced with their own group.

Algorithm 2 shows the reduce function in details. Given two lists of subgraph clones under the same repository function f_s, the reduce function compares their subgraph clones (Lines 1 and 2) and checks if two graphs can be connected (Lines 4 to 13) by referring to the the control flow graph edges E_s and E_t . If the two subgraph clones can be connected by one clone pair, then they can be merged into a single one (Lines 6 and 7). If a subgraph clone from sg_s^2 cannot be merged into any subgraph clones in sg_s^2, it will be appended to the list sg_s^1 (Lines 20 and 21). At the end of the

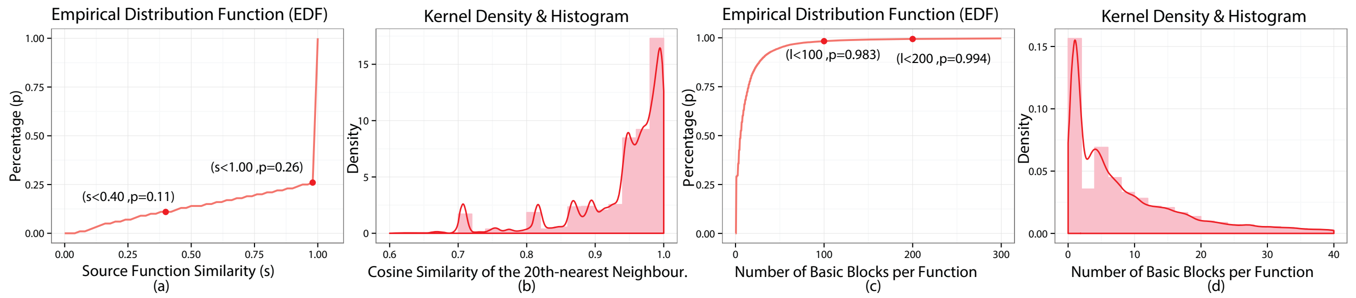


Figure 6: (a) the EDF function on repository function clone pair similarity, (b) the kernel density & histogram of the cosine similarity of each basic block's 20th-nearest neighbor, (c) the EDF on per assembly function block count $|B_s|$, and (d) the kernel density & histogram on assembly function block count $|B_s| < 40$.

Algorithm 2 Reducer

Input subgraph lists of same f_s: $sg_s^1[1:a_1]$ and $sg_s^2[1:a_2]$
Output a single subgraph list $sg_s^1[1:a]$

```
 1: for $a_1 \to |sg_s^1|$ do
 2:     for $a_2 \to |sg_s^2|$ do
 3:         $canMerge \leftarrow false$
 4:         for each $\langle b_t^{a1}, b_s^{a1} \rangle \in sg_s^1[a_1]$ do
 5:             for each $\langle b_t^{a2}, b_s^{a2} \rangle \in sg_s^2[a_2]$ do
 6:                 if $E_t(b_t^{a1}, b_t^{a2})$ exists then
 7:                     if $E_s(b_s^{a1}, b_s^{a2})$ exists then
 8:                         $canMerge \leftarrow true$
 9:                         goto Line 14.
10:                     end if
11:                 end if
12:             end for
13:         end for
14:         if $canMerge$ is $true$ then
15:             $sg_s^1[a_1] \leftarrow sg_s^1[a_1] \bigcup sg_s^2[a_2]$
16:             $sg_s^2 \leftarrow sg_s^2 - sg_s^2[a_2]$
17:         end if
18:     end for
19: end for
20: if $sg_s^2$ is not $\emptyset$ then    ▷ for graphs in $sg_s^2$ that cannot be merged, append them to the list
21:     $sg_s^1 \leftarrow sg_s^1 \bigcup sg_s^2$
22: end if
23: return $sg_s^1$
```

graph search algorithm, we solve the problem in Definition 1. In order to obtain a ranked list of repository functions for f_s, we calculate the similarity value by checking how much its subgraphs sg_s cover the graph of the query f_t: $sim_s = (|\text{uniqueEdges}(sg_s)| + |\text{uniqueNodes}(sg_s)|)/(|B_t| + |E_t|)$.

Compared to other join-based or graph-exploration-based search approach, our MapReduce-based search procedure avoids recursive search and is bounded by polynomial complexity. Let m_s be the number of clone pairs for a target function f_t. There are at most $O(m_s^2)$ connectivity checks between the clone pairs (no merge can be found) and the map function requires $O(m_s)$ executions. m_s corresponds to the number of rectangles in the second row of Figure 5. Refer from the second row to the last one. The reduce function are bounded by $O(m_s^2)$ comparisons. m_s is bounded by $O(|B_t| \times |B_s|)$, which implies that each basic block of f_t is a clone with all the basic blocks of f_s. However, this extreme case rarely happens. Given the nature of assembly functions and search scenarios, m_s is sufficiently bounded by $O(max(|B_t|, |B_s|))$. According to the descriptive statistics of our experiment, 99% of them have less than 200 basic blocks.

8. EXPERIMENTS

This section presents comprehensive experimental results for the task of assembly code clone search. First, we explain how to construct a labeled dataset that can be used for benchmarking in future research. Then, we evaluate the effect of assembly code normalization. Although normalization has been extensively used in previous works, its effects have not been thoroughly studied yet. Next, we present the benchmark results which compares Kam1n0 with state-of-the-art clone search approaches in terms of clone search quality. Finally, we demonstrate the scalability and capacity of the Kam1n0 engine by presenting the experimental results from a mini-cluster.

Library Name	Branch Count	Function Count	Block Count	Clone Pair Count
bzip2	5	590	15,181	1,329
curl	16	9,468	176,174	49,317
expat	3	2,025	35,801	14,054
jsoncpp	14	11,779	43,734	204,701
libpng	9	4,605	82,257	18,946
libtiff	13	7,276	124,974	51,925
openssl	9	13,732	200,415	29,767
sqlite	12	9,437	202,777	23,674
tinyxml	7	3,286	30,401	22,798
zlib	8	1,741	30,585	6,854
total	96	63,939	942,299	423,365

Table 1: The assembly code clone dataset summary.

8.1 Labeled Dataset Generation

One of the challenging problems for assembly code clone search is the lack of labeled (ground truth) dataset, since the most effective labeled dataset requires intensive manual identification of assembly code clones [7]. To facilitate future studies on assembly clone search, we have developed a tool to systematically generate the assembly function clones based on source code clones. The tool performs four steps: *Step 1:* Parse all the source code functions from different branches or versions of a project and identify all the function-to-function clones using CCFINDERX [16], which estimates source code similarity based on the sequence of normalized tokens. *Step 2:* Compile the given branches or versions of a project with an additional debug flag to enable the compiler output debug symbols. *Step 3:* Link the source code functions to the assembly code functions using the compiler output debug symbols, where such information is available. *Step 4:* For each pair of source code clone, generate a pair of assembly function clone and transfer the similarity to the new pair.

The intuition is that the source code function-level clones indicate the functional clones between their corresponding assembly code. In [7, 25], the source code and assembly code are manually linked with an injected identifier in the form of variable declarations. However, after the automation of such process, we find that the link rate is very low due to the impact of compiler optimizations. The generated

M	Approach	Bzip2	Curl	Expat	Jsoncpp	Libpng	Libtiff	Openssl	Sqlite	Tinyxml	Zlib	Avg.
	BinClone	.985	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	***.894**	.188
	Composite	.857	.766	.693	.725	.814	.772	.688	.726	.688	.729	.746
	Constant	.769	.759	.723	.665	.829	.764	.689	.776	.683	.768	.743
	Graphlet	.775	.688	.673	.563	.714	.653	.682	.746	.676	.685	.685
A	Graphlet-C	.743	.761	.705	.604	.764	.729	.731	.748	.677	.668	.713
U	Graphlet-E	.523	.526	.505	.516	.519	.521	.512	.513	.524	.514	.517
R	MixGram	.900	.840	.728	.726	.830	.808	.809	.765	.707	.732	.785
O	MixGraph	.769	.733	.706	.587	.755	.692	.713	.765	.674	.708	.710
C	N-gram	.950	.860	.727	.713	.843	.809	.819	.789	.714	.766	.799
	N-perm	.886	.847	.731	.729	.834	.813	.811	.769	.709	.736	.787
	Tracelet	.830	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.799	.163
	LSH-S	.965	.901	.794	.854	.894	.922	.882	.845	.768	.758	.858
	Kam1n0	***.992**	***.989**	***.843**	***.890**	***.944**	***.967**	***.891**	***.895**	***.864**	.830	***.911**
	BinClone	.294	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.091	.038
	Composite	.645	.495	.375	.353	***.541**	.482	.288	.405	.261	.409	.425
	Constant	.247	.280	.301	.158	.311	.349	.072	.157	.142	.240	.226
	Graphlet	.162	.133	.138	.051	.115	.103	.041	.108	.150	.106	.111
A	Graphlet-C	.455	.482	.296	.176	.413	.369	.366	.437	.245	.338	.358
U	Graphlet-E	.022	.024	.013	.020	.012	.015	.004	.010	.020	.026	.017
P	MixGram	.727	.598	.363	.337	.513	.512	***.464**	.471	.286	.383	.465
R	MixGraph	.247	.242	.228	.098	.196	.184	.078	.180	.163	.175	.179
	N-gram	.638	.491	.297	.275	.408	.428	.301	.417	.264	.314	.383
	N-perm	.613	.589	.360	.344	.523	***.515**	.438	.465	.288	.370	.450
	Tracelet	.057	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.027	.008
	LSH-S	.227	.014	.095	.049	.035	.038	.012	.018	.079	.041	.061
	Kam1n0	***.780**	***.633**	***.473**	***.504**	.477	.387	.411	***.610**	***.413**	***.465**	***.515**
	BinClone	.495	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.398	.089
	Composite	.505	.525	.489	.190	.493	.536	.238	.382	.303	.472	.413
	Constant	.354	.459	.539	.132	.473	.502	.199	.379	.229	.495	.376
	Graphlet	.274	.309	.408	.030	.264	.276	.154	.303	.233	.302	.255
M	Graphlet-C	.339	.499	.586	.084	.412	.449	.284	.416	.272	.361	.370
A	Graphlet-E	.021	.053	.010	.011	.024	.040	.012	.019	.039	.028	.026
P	MixGram	.559	.641	.625	.191	.511	.589	.392	.445	.321	.474	.475
@	MixGraph	.334	.407	.572	.064	.345	.351	.211	.387	.244	.371	.329
10	N-gram	.620	.636	.615	.176	.512	.567	.398	.481	.310	.506	.482
	N-perm	.532	.653	.628	.191	.516	.597	.394	.452	.317	.483	.476
	Tracelet	.228	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.265	.049
	LSH-S	.322	.069	.198	.032	.145	.078	.086	.111	.130	.101	.127
	Kam1n0	***.672**	***.680**	***.690**	***.196**	***.548**	***.587**	***.434**	***.605**	***.375**	***.573**	***.536**

Table 2: Benchmark results of different assembly code clone search approaches. We employed three evaluation metrics: the *Area Under the Receiver Operating Characteristic Curve* (*AUROC*), the *Area Under the Precision-Recall Curve* (*AUPR*), and the *Mean Average Precision at Position 10* (*MAP@10*). Ø denotes that the method is not scalable and we cannot obtain a result for this dataset within 24 hours.

assembly code clone is in fact the combined result of source code patches and compiler optimizations. The source code evolves from version to version, and different versions may have different default compiler settings. Thus, the labeled dataset simulates the real-world assembly clone evolvement. This tool is applicable only if the source code is available.

Refer to Table 1 for some popular open source libraries with different versions. We applied the aforementioned tool on them to generate the labeled dataset for the experiments. There are 63,939 assembly functions which successfully link to the source code functions. The labeled dataset is a list of one-to-multiple assembly function clones with the transferred similarity from their source code clones.

See Figure 6c and Figure 6d. The assembly function basic block count follows a long-tail distribution. Most of them have between 0 and 5 assembly basic blocks, and 99% of them is bounded by 200. We find that this is the typical distribution of assembly function block count. This distribution facilitates our graph search because the worst case is bounded by $O(|B_t| \times |B_s|)$ and $P[|B_s| < 200] > 0.99$. Figure 6b shows the cosine similarity distribution of each basic block's 20th-nearest neighbor. It reflects variations of density in the vector space and calls for an adaptive LSH.

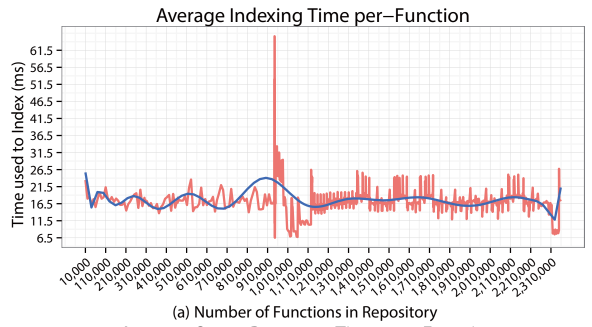
74% of the source code clones given by CCFINDERX are exact clones (see Figure 6a). However, by applying a strong hash on their assembly code, we find that only 30% of them are exact clones (Type I clones). Thus, the total percentage of inexact clones is 70%×74%+26% = 77.8%. CCFINDERX classifies tokens in source code into different types before clone detection. If two source code fragments are identified as clones with a low similarity, there is a higher chance that the underlying assembly code is indeed not a clone due to the normalization of the source code. To mitigate this issue, we heuristically set a 0.4 threshold for clones to be included in our dataset. Thus, we have 66.8% of inexact clones.

8.2 Normalization Level

	None	Root	Specific
Root	$< 2e^{-16}$	-	-
Specific	$< 2e^{-16}$	1	-
Type	$< 2e^{-16}$	1	1

Table 3: Paired t-test on the normalization level.

Assembly code normalization is used in [7, 27]. However, its effects were not formally studied. In this section, we present the results of the statistical test on the effects of the normalization level. Details on normalization can be found in our technical report.[6] We start by using a strong hash clone search with different normalization levels on each of the generated datasets. Then, we collect the corresponding precision value to the given normalization level as samples and test the relationship between precision and the chosen normalization level. Normalization can increase the recall, but we want to evaluate the trade-off between the precision and different normalization levels. According to the ANOVA



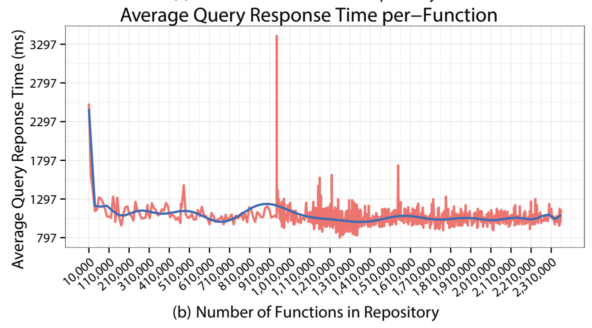


Figure 7: Scalability study. (a): Average Indexing Time vs. Number of Functions in the Repository. (b): Average Query Response Time vs. Number of Functions in the Repository. The red line represents the plotted time and the blue line represents the smoothed polynomial approximation.

test ($p < 2e^{-16}$), the difference of applying normalization or not is statistically significant. Consider Table 3. However, the difference of applying different levels, namely *Root*, *Type*, or *Specific*, is not statistically significant.

8.3 Clone Search Approach Benchmark

In this section, we benchmark twelve assembly code clone search approaches: BinClone [8, 19], Graphlets [17, 18], LSH-S [29], and Tracelet [6]. [18] includes several approaches: 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 Mix-Graph), 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 [20]. We re-implemented all these approaches under a unified evaluation framework and all parameters were configured according to the papers. We did not include the re-write engine in [6] because it is not scalable.

Several metrics are used in previous research to evaluate the clone search quality, but there is no common agreement on what should be used. Precision, recall and F1 are used in BinClone [7], while [36] maintains that a F2 measure is more appropriate. However, these two values will change as the search similarity threshold value changes. To evaluate the trade-off between recall and precision, we use three typical information retrieval metrics, namely *Area Under the Receiver Operating Characteristic Curve* (*AUROC*), *Area Under the Precision-Recall Curve* (*AUPR*), and *Mean Average Precision at Position 10* (*MAP@10*). These three metrics flavor different information retrieval scenarios. Therefore, we employ all of them. AUROC and AUPR can test a classifier by issuing different threshold values consecutively [9, 24, 30], while MAP@10 can evaluate the quality of the top-ranked list simulating the real user experience [23].

Table 2 presents the benchmark results. The highest score of each evaluation metric is highlighted for each dataset. Also, the micro-average of all the results for each approach is given in the *Avg* column. Kam1n0 out-performs the other approaches in almost every case for all evaluation metrics. Kam1n0 also achieves the best averaged AUROC, AUPRC, and MAP@10 scores. The overall performance also suggests that it is the most stable one. Each approach is given a 24-hour time frame to finish the clone search and it is only allowed to use a single thread. Some results for BinClone and Tracelet are empty, which indicate that they are not scalable enough to obtain the search result within the given time frame. Also, we notice that BinClone consumes more memory than the others for building the index, due to its combination of features which enlarges the feature space. We notice that the experimental results are limited within the context of CCFinder. In the future, we will investigate other source code clone detection techniques to generate the ground truth data.

8.4 Scalability Study

In this section, we evaluate Kam1n0's scalability on a large repository of assembly functions. We set up a mini-cluster on *Google Cloud* with four computational nodes. Each of them is a *n1-highmem-4* machine with 2 virtual cores and 13 GB of RAM. We only use regular persistent disks rather than solid state drives. Each machine is given 500 GB of disk storage. All the machines run on *CentOS*. Three machines run the *Spark Computation Framework* and the *Apache Cassandra Database* and the other runs our Kam1n0 engine. To conduct the experiment, we prepare a large collection of binary files. All these files are either open source libraries or applications, such as *Chromium*. In total, there are more than 2,310,000 assembly functions and 27,666,692 basic blocks. Altogether, there are more than 8 GB of assembly code. We gradually index this collection of binaries in random order, and query the zlib binary file of version 2.7.0 on Kam1n0 at every 10,000 assembly function indexing interval. As zlib is a widely used library, it is expected that it has a large number of clones in the repository. We collect the average indexing time for each function to be indexed, as well as the average time it takes to respond to a function query. Figure 7 depicts the average indexing and query response time for each function. The two diagrams suggest that Kam1n0 has a good scalability with respect to the repository size. Even as the number of functions in the repository increases from 10,000 to 2,310,000, the impact on the response time is negligible. There is a spike up at 910,000 due to the regular compaction routine in Cassandra, which increases I/O contention in the database.

9. CONCLUSION AND LESSON LEARNED

Through the collaboration with Defence Research and Development Canada (DRDC), we learned that scalability, which was not considered in previous studies, is a critical issue for deploying a successful assembly clone search engine. To address this, we present the first assembly search engine that combines LSH and subgraph search. Existing off-the-shelf LSH nearest neighbor algorithms and subgraph isomorphism search techniques do not fit our problem setting. Therefore, we propose new variants of LSH scheme and incorporate it with graph search to address the challenges. Experimental results suggest that our proposed MapReduce-based system, Kam1n0, is accurate, efficient, and scalable. Currently Kam1n0 can only identify clones for x86/amd64 processor. In the future, we will extend it to the other processors and investigate approaches that can find clones between different processors. Kam1n0 provides a practical solution of assembly clone search for both DRDC and the reverse engineering community. The contribution is partially reflected by the award received at the 2015 Hex-Rays Plug-In Contest.

10. ACKNOWLEDGMENT

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) and Canada Research Chairs Program (950-230623).

11. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1), 2008.

[2] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *Proc. of WWW'05*, 2005.

[3] M. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings on 34th Annual ACM STOC'02*, 2002.

[4] P. Charland, B. C. M. Fung, and M. R. Farhadi. Clone search for malicious code correlation. In *Proc. of the NATO RTO Symposium on Information Assurance and Cyber Defense (IST-111)*, 2012.

[5] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. of ACM SoCG'04*, 2004.

[6] Y. David and E. Yahav. Tracelet-based code search in executables. In *Proc. of SIGPLAN'14*, 2014.

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

[8] M. R. Farhadi, B. C. M. Fung, Y. B. Fung, P. Charland, S. Preda, and M. Debbabi. Scalable code clone search for malware analysis. *Digital Investigation*, 2015.

[9] T. Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8), 2006.

[10] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proc. of SIGMOD'12*, 2012.

[11] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi. Dsh: Data sensitive hashing for high-dimensional k-nnsearch. In *Proc. of SIGMOD'14*. ACM, 2014.

[12] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of the VLDB*, 1999.

[13] W. Han, J. Lee, and J. Lee. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. of SIGMOD'13*, 2013.

[14] S. Har-Peled, P. Indyk, and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of Computing*, 8(1), 2012.

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

[16] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE TSE*, 28(7), 2002.

[17] W. M. Khoo. Decompilation as search. *University of Cambridge, Computer Laboratory, Technical Report*, 2013.

[18] W. M. Khoo, A. Mycroft, and R. J. Anderson. Rendezvous: a search engine for binary code. In *Proc. of MSR'13*, 2013.

[19] V. Komsiyski. Binary differencing for media files. 2013.

[20] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proc. of RAID'06*. Springer, 2006.

[21] J. Lee, W. Han, R. Kasperovics, and J. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2), 2012.

[22] Y. Liu, J. Cui, Z. Huang, H. Li, and H. T. Shen. SK-LSH: an efficient index structure for approximate nearest neighbor search. *PVLDB*, 7(9), 2014.

[23] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge University Press, 2008.

[24] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing.* MIT press, 1999.

[25] A. Mockus. Large-scale code reuse in open source software. In *Proc. of FLOSS'07*. IEEE, 2007.

[26] A. Saebjornsen. *Detecting Fine-Grained Similarity in Binaries*. PhD thesis, UC Davis, 2014.

[27] A. Sæbjørnsen, J. Willcock, T. Panas, D. J. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proc. of the 18th International Symposium on Software Testing and Analysis*, 2009.

[28] R. Shamir and D. Tsur. Faster subtree isomorphism. In *Proc. of IEEE ISTCS'97*, 1997.

[29] M. Sojer and J. Henkel. Code reuse in open source software development: Quantitative evidence, drivers, and impediments. *JAIS*, 11(12), 2010.

[30] S. Sonnenburg, G. Rätsch, C. Schäfer, and B. Schölkopf. Large scale multiple kernel learning. *JMLR*, 7, 2006.

[31] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *The VLDB Endowment*, 5(9), 2012.

[32] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *Proc. of SIGMOD'09*, 2009.

[33] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM TODS*, 35(3), 2010.

[34] J. R. Ullmann. An algorithm for subgraph isomorphism. *ACM JACM*, 23(1), 1976.

[35] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *arXiv:1408.2927*, 2014.

[36] H. Welte. Current developments in GPL compliance, 2012.