# Kam1n0

## Assembly Code Search Engine for Reverse Engineer

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

*Data Mining and Security Lab, School of Information Studies, McGill University, Montreal, Canada

+Mission Critical Cyber Security Section, Defence R&D Canada – Valcartier, Quebec, Canada

## 1 Research Problem

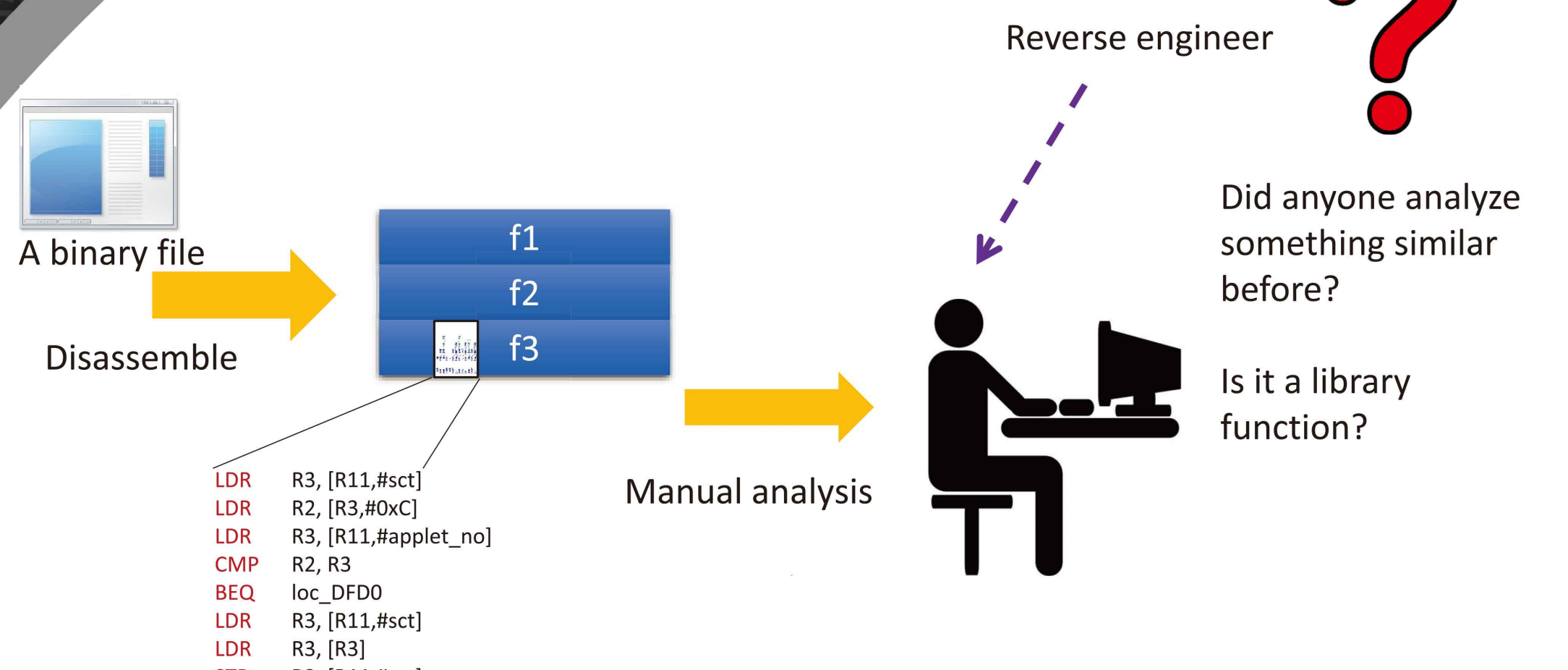
A binary file
Disassemble
f1
f2
f3
Reverse engineer
Did anyone analyze something similar before?
Is it a library function?
Manual analysis

Figure 1. The bianry analysis process

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.

A binary file can be disassembled in to as list of assembly functions. A function can be represented as a *control flow graph* (see Figure 2). Per our discussion with 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. A subgraph consists of several interconnected *basic blocks*; and there are three types of clones between basic blocks.

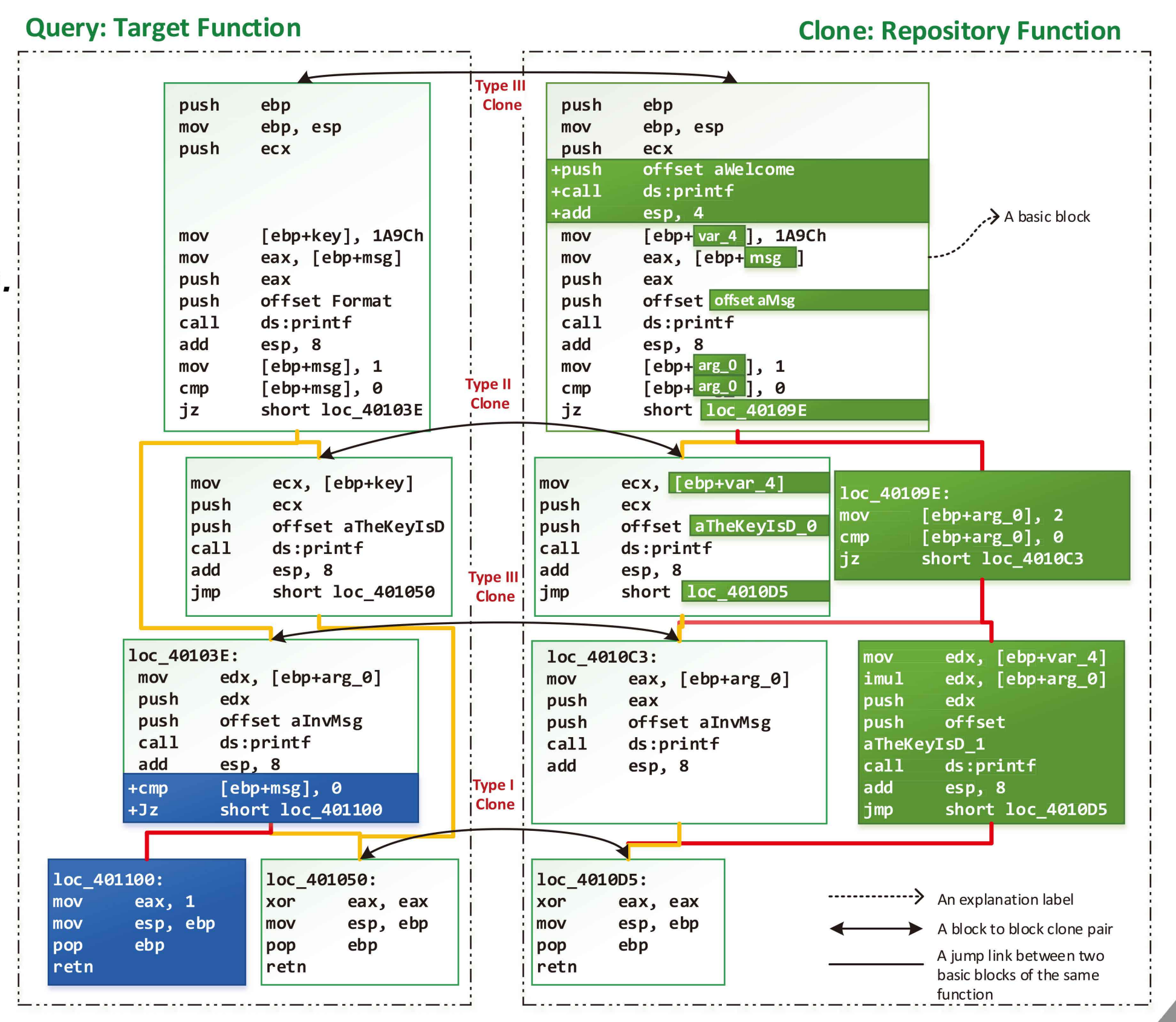
Query: Target Function
Clone: Repository Function
Type II Clone
Type I Clone
Type III Clone
A basic block
An explanation label
A block-to-block clone pair
A jump link between two basic blocks of the same function

Figure 2. An example cloned subgraph

## 2 Overall Architecture

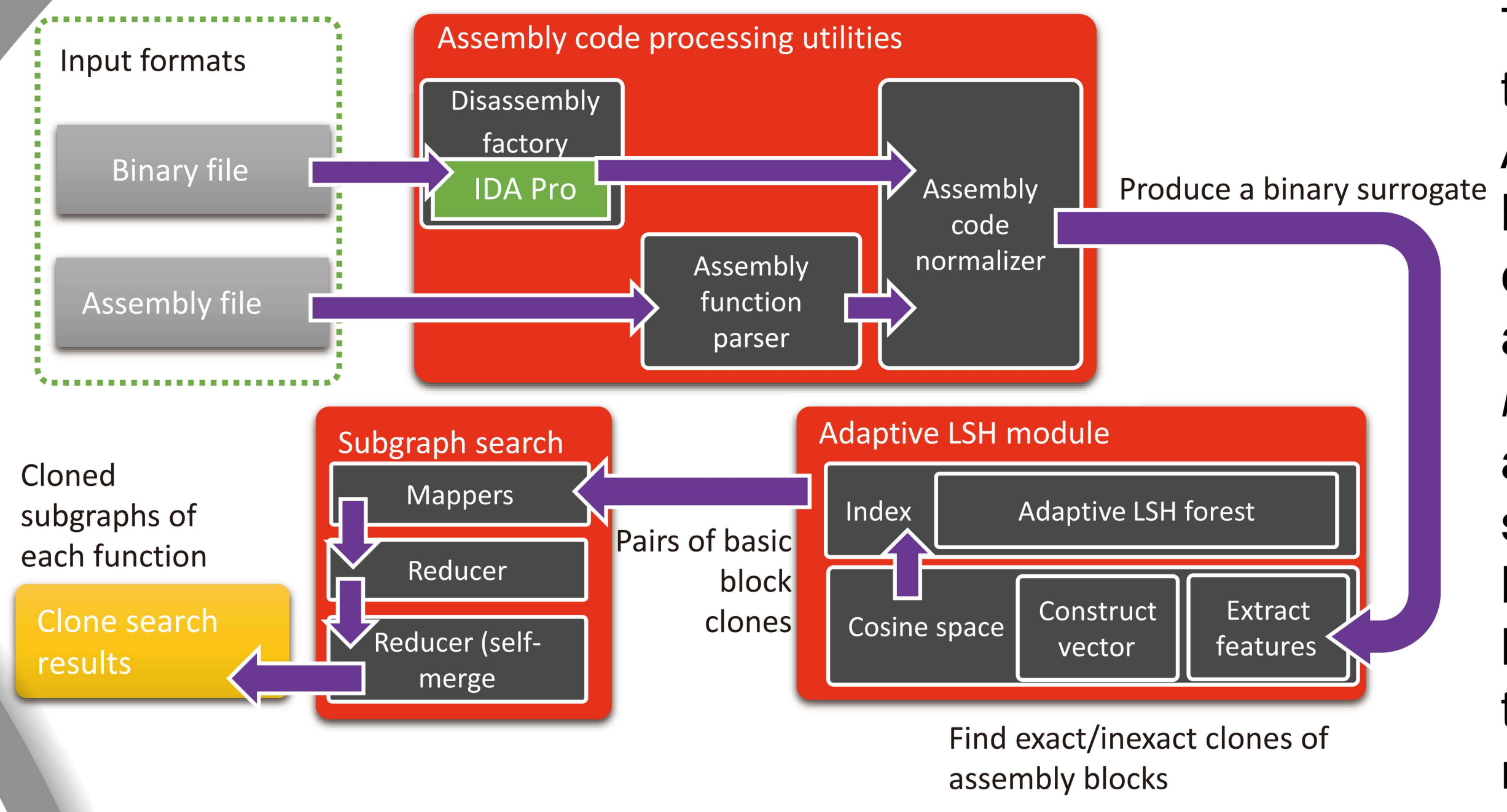

Figure 3. Assembly clone search data flow.

The search process consists of the three 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. *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 adaptive locality sensitive hashing. *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.

The Kam1n0 engine is designed for general key-value storage and the Apache Spark 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 engine. The Kam1n0 engine splits a search query into multiple jobs and coordinates their execution flow.

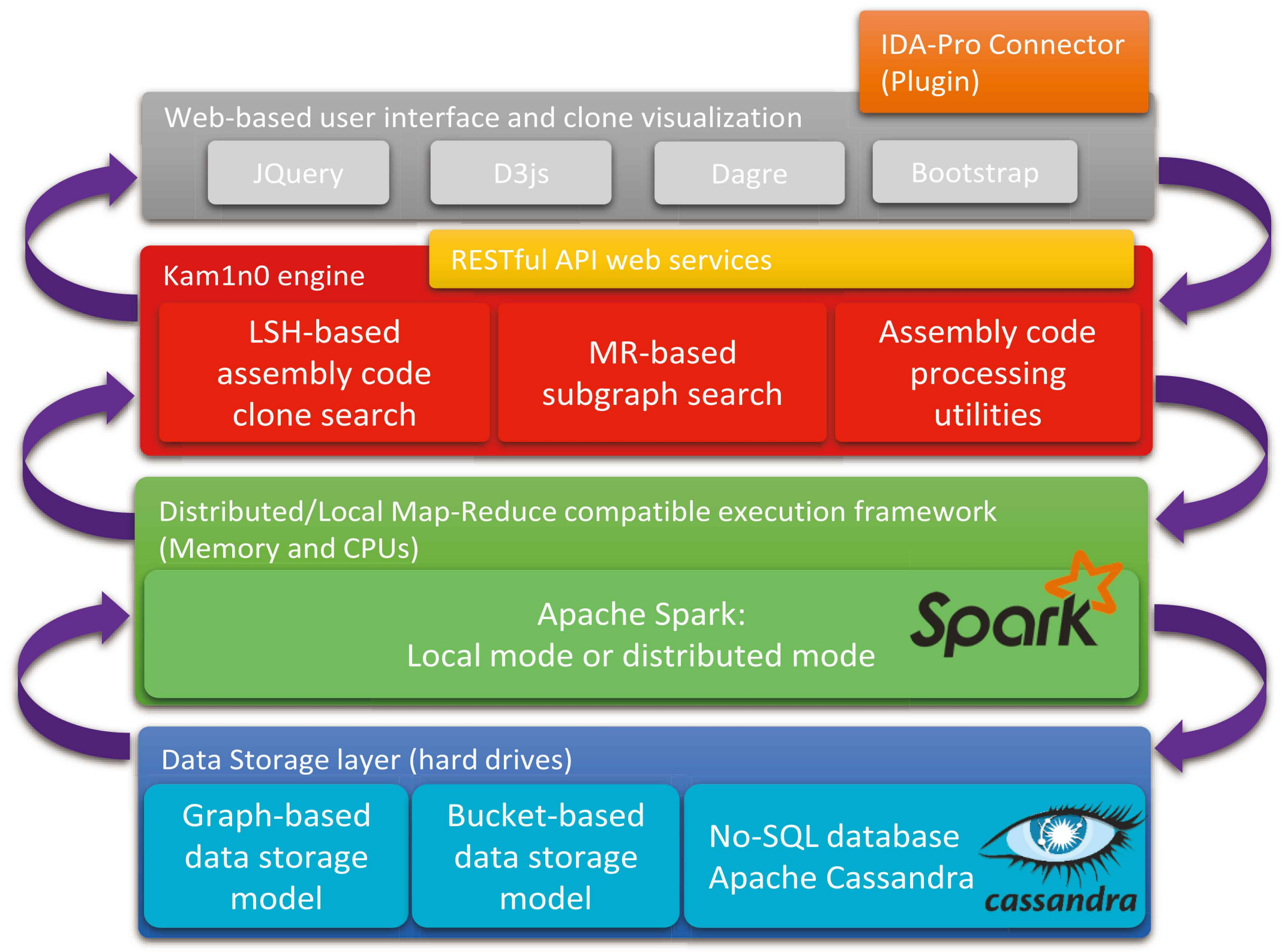
IDA-Pro Connector (Plugin)
Web-based user interface and clone visualization
JQuery
D3js
Dagre
Bootstrap
Kam1n0 engine
RESTful API web services
LSH-based assembly code clone search
MR-based subgraph search
Assembly code processing utilities
Distributed/Local Map-Reduce compatible execution framework (Memory and CPUs)
Apache Spark: Local mode or distributed mode
Spark
Data Storage layer (hard drives)
Graph-based data storage model
Bucket-based data storage model
No-SQL database Apache Cassandra
cassandra

Figure 4. The solution stack of implementation.

Kam1n0 is open-source and available on GitHub. Scan the code to check out Kam1n0 and the paper.

## 3 Technical Details

*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. ALSH organizes buckets into a tree structure. To our best knowledge, ALSH is the First incremental locality sensitive hashing scheme that solves this issue specifically for cosine space with theoretical guarantee.

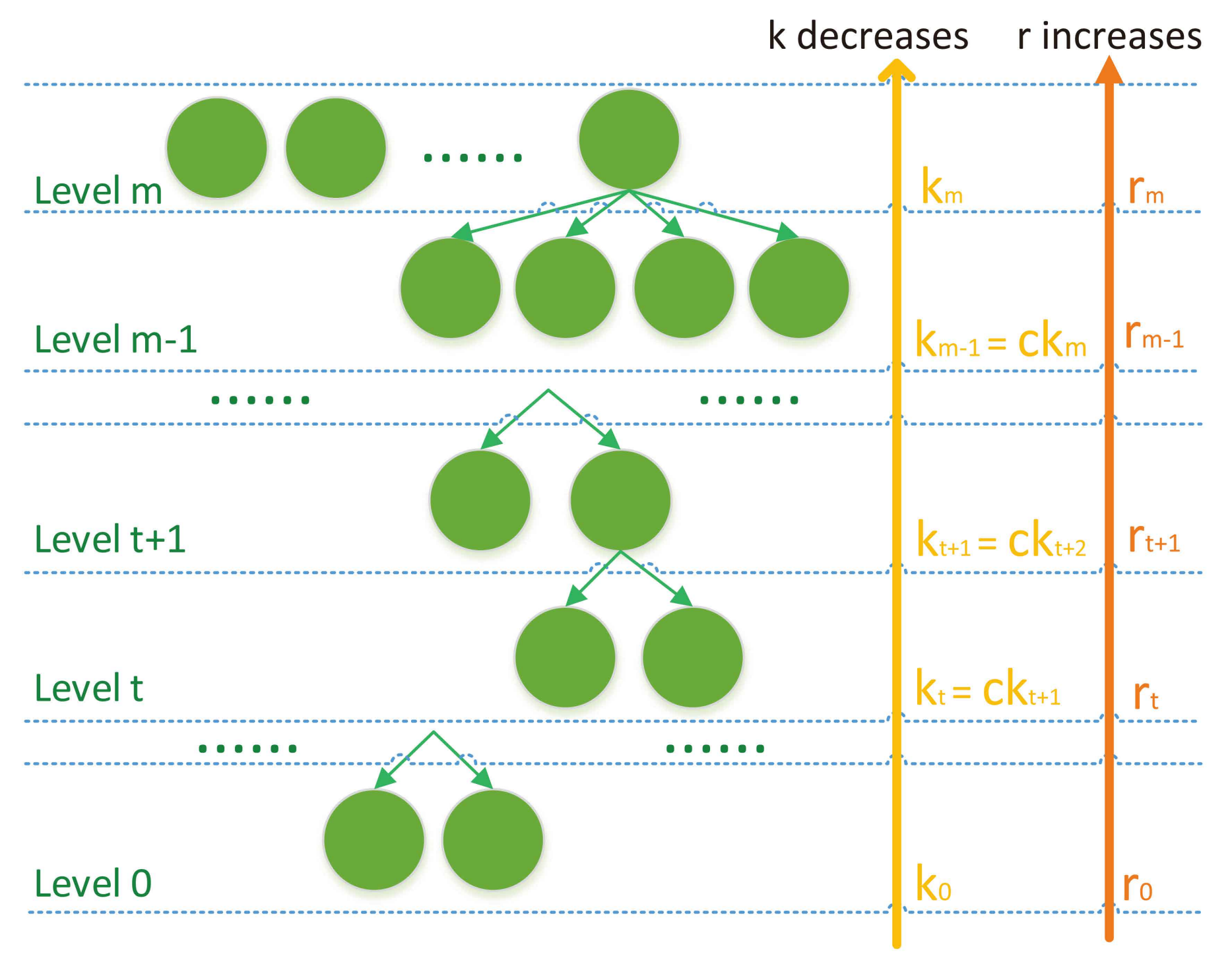

Figure 5. The index structure for the Adaptive Locality Sensitive Hashing (ALSH).

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.

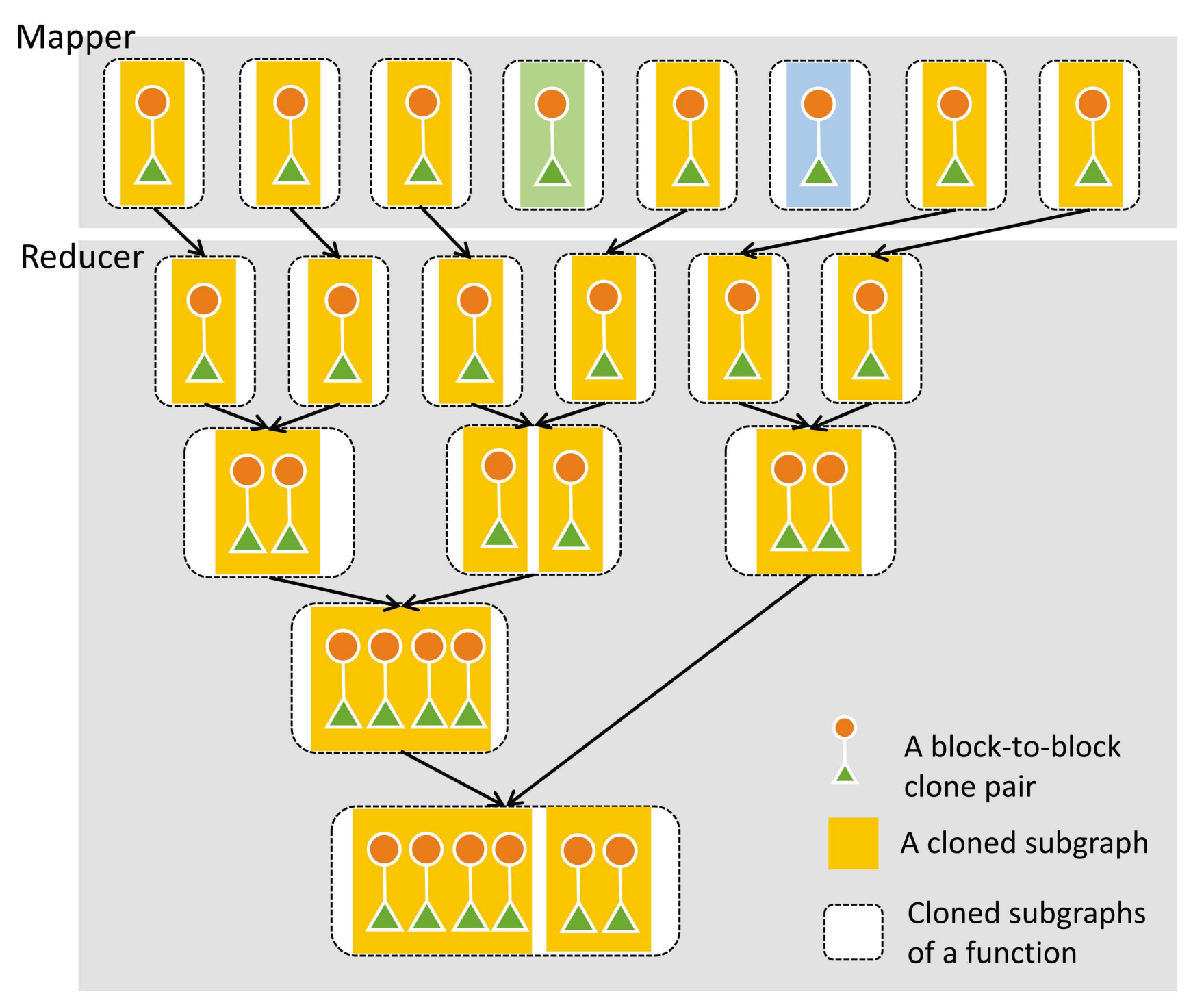

Figure 6. The index structure for the Adaptive Locality Sensitive Hashing (ALSH).

## 4 Experimental Evaluation

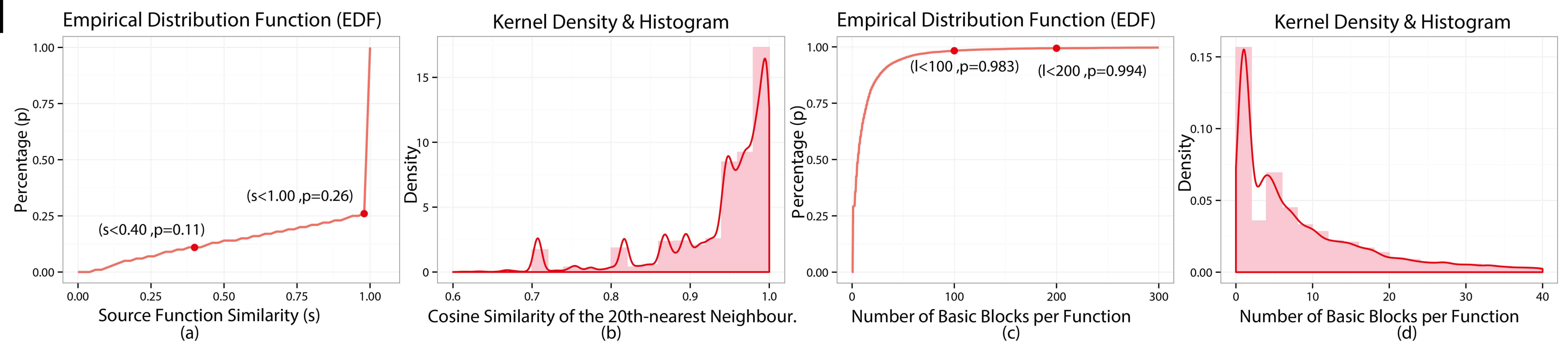

Figure 7: (a) the EDF function on repository function clone pair similarity, (b) the kernel density & histogram of the cosine similarity of each basic block's 20th-nearest neighbor, (c) the EDF on per assembly function block count, and (d) the kernel density & histogram on assembly function block count < 40.

We 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 the performance of twelve existing state-of-the-art solutions on the dataset using three typical information retrieval metrics (Table 1). Kam1n0 boosts the clone search quality and yields stable results across different datasets and metrics.

We also setup a mini-cluster (4 nodes) to evaluate the scalability of Kam1n0 (Figure 8).

| M | Approach | Bzip2 | Curl | Expat | Jsoncpp | Libpng | Libtiff | Openssl | Sqlite | Tinyxml | Zlib | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AUROC | 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 |
| | Graphlet-C | .743 | .761 | .705 | .604 | .764 | .729 | .731 | .748 | .677 | .668 | .713 |
| | Graphlet-E | .523 | .526 | .505 | .516 | .519 | .521 | .512 | .513 | .524 | .514 | .517 |
| | MixGram | .900 | .840 | .728 | .726 | .830 | .808 | .809 | .765 | .707 | .732 | .785 |
| | MixGraph | .769 | .733 | .706 | .587 | .755 | .692 | .713 | .765 | .674 | .708 | .710 |
| | *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** |
| AUPR | 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 |
| | Graphlet-C | .455 | .482 | .296 | .176 | .413 | .369 | .366 | .437 | .245 | .338 | .358 |
| | Graphlet-E | .022 | .024 | .013 | .020 | .012 | .015 | .004 | .010 | .020 | .026 | .017 |
| | MixGram | .727 | .598 | .363 | .337 | .513 | .512 | ***.464** | .471 | .286 | .383 | .465 |
| | 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** |
| MAP@10 | 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 |
| | Graphlet-C | .339 | .499 | .586 | .084 | .412 | .449 | .284 | .416 | .272 | .361 | .370 |
| | Graphlet-E | .021 | .053 | .010 | .011 | .024 | .040 | .012 | .019 | .039 | .028 | .026 |
| | MixGram | .559 | .641 | .625 | .191 | .511 | .589 | .392 | .445 | .321 | .474 | .475 |
| | MixGraph | .334 | .407 | .572 | .064 | .345 | .351 | .211 | .387 | .244 | .371 | .329 |
| | *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 1: 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.**

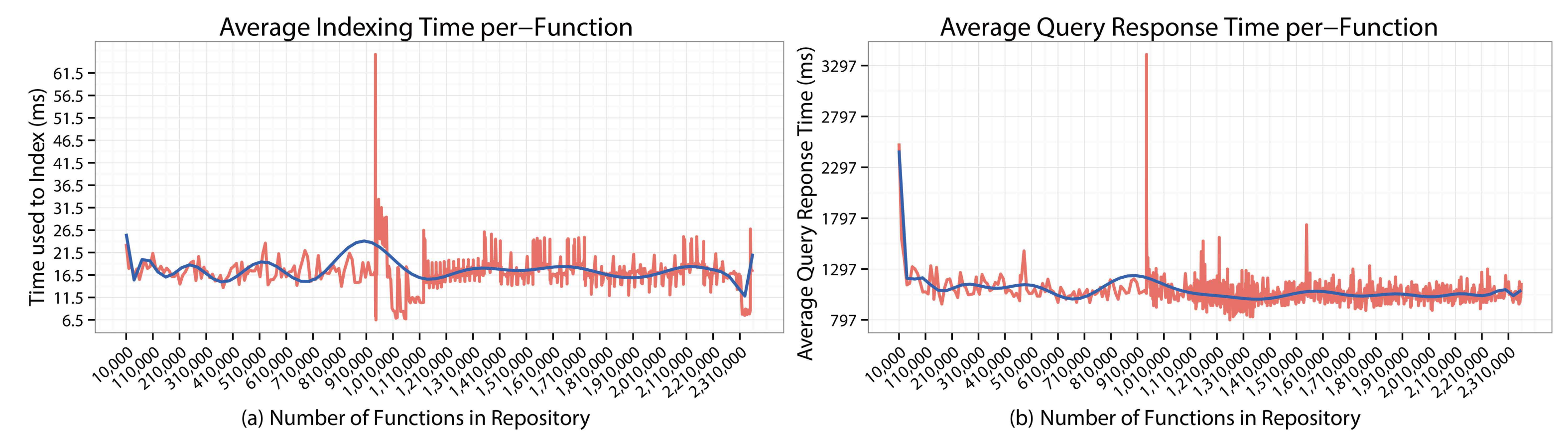

Figure 8. Scalability study. (a): Average Indexing Time vs. Number of Functions in the Repository. (b): Average Query Response Time vs. Number of Functions in the Repository.