BCD: Decomposing Binary Code Into Components Using Graph-Based Clustering

Vishal Karande
The University of Texas at Dallas
vishal.karande@utdallas.edu

Swarup Chandra
The University of Texas at Dallas
swarup.chandra@utdallas.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Juan Caballero
IMDEA Software Institute
juan.caballero@imdea.org

Latifur Khan
The University of Texas at Dallas
lkhan@utdallas.edu

Kevin Hamlen
The University of Texas at Dallas
hamlen@utdallas.edu

ABSTRACT

Complex software is built by composing components implementing largely independent blocks of functionality. However, once the sources are compiled into an executable, that modularity is lost. This is unfortunate for code recipients, for whom knowing the components has many potential benefits, such as improved program understanding for reverse-engineering, identifying shared code across different programs, binary code reuse, and authorship attribution. This paper proposes a novel approach for decomposing such source-free program executables into components. Given an executable, our approach first statically builds a decomposition graph, where nodes are functions and edges capture three types of relationships: code locality, data references, and function calls. It then applies a graph-theoretic approach to partition the functions into disjoint components. A prototype implementation, BCD, demonstrates the approach's efficacy: Evaluation of BCD with 25 C++ binary programs to recover the methods belonging to each class achieves high precision and recall scores for these tested programs.

CCS CONCEPTS

• **Information systems** → *Clustering*; • **Software and its engineering** → *Automated static analysis*;

KEYWORDS

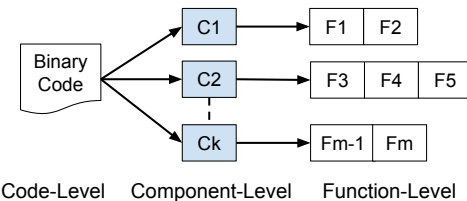Binary code decomposition, Components, Graph-Based Clustering

Figure 1: Illustration of binary code hierarchy.

1 INTRODUCTION

Complex software is built by composing smaller *components* that implement largely independent blocks of functionality. For example, Figure 1 illustrates a hierarchy of an executable file that contains m functions, denoted Fx with $x \in [1, m]$. These functions can be associated with k modules or components, denoted Cx with $x \in [1, k]$. Such components are integral to widely-used programming paradigms like *modular programming* and *object-oriented programming*. For instance, each class in a C++ program can be considered a separate component. Even in programming languages like C that lack classes, modules, and packages, programmers often place each component in its own source file and provide interfaces through header files. Such modular software design is key to keeping code complexity at bay, controlling code development and maintenance costs, and facilitating code reuse.

Once the source files are compiled into object files and those object files are statically linked into an executable, this structural modularity is hidden. This is unfortunate because most commercial off-the-shelf (COTS) software are released as executables without debugging information. Binary code analysis performed over a third-party executable (without access to its source code) could greatly benefit from modularity information. Security applications that could benefit include program understanding and decompiling [4, 11], finding related functions like the decryption routine for a given encryption routine [6], identifying shared code across different programs [16, 23, 29, 32, 33], reusing binary code [5, 18], authorship attribution [1, 20, 26], and binary-level enforcement of object flow integrity policies [31]. In all these applications, analysis of an unknown binary at only the function level is time consuming or inadequate, especially when the number of functions is large. However, it may be intuitively easier or more effective to analyze functions that are grouped at the component level.

For instance, many binary code reuse applications entail reusing a set of functions belonging to a program component as a unit,

rather than reusing the functions individually [5, 17, 18]. Intuitively, component reuse is more useful since a set of functions (e.g., a library) can handle more complex logical tasks than individual functions. As another example, consider the application of authorship attribution using machine learning approaches [1, 26]. Discovery of discriminatory stylistic patterns of code authors is potentially enhanced by knowledge of each program's components, since patterns within each isolated function might be too fine-grained and dispersed, whereas patterns spanning component-related function families may provide a richer feature set.

This paper presents a novel static approach to decompose an executable into components. Our approach is independent of the compiler used to generate the executable and applies to both object-oriented and procedural programs. The main idea in our approach is to examine three key properties (code locality, data references and function calls) that preserve useful information regarding the original program components throughout the compilation process. We represent these properties in the form of a *decomposition graph*, and use a graph-theoretic clustering algorithm to identify the components. We have implemented our approach as a tool called BCD. Our empirical evaluation of BCD on 25 C++ programs, whose ground truth we manually extracted using source code and debugging symbols, shows high component detection accuracy.

2 BINARY CODE DECOMPOSITION

While much structural information is lost during compilation, an executable still maintains useful information that can aid in identifying program components created by the programmers or introduced by a programming paradigm. In this section, we first describe how BCD builds a graph for each of the three key decomposition properties of code locality, data references, and function calls (Section 2.1). We then detail how BCD builds the decomposition graph from the three property graphs (Section 2.2). Finally, we describe the clustering algorithm to partition the decomposition graph into components (Section 2.3).

2.1 Decomposition Properties

The first step in our approach is to build a directed graph for each of the three key decomposition properties of code locality, data references, and function calls. In each graph, nodes correspond to the functions in the executable. Different tools can be used to identify functions in an executable such as IDA [8], BYTEWEIGHT [3], and Dyninst [14]. BCD currently uses IDA to identify the functions in an executable, but can easily be adapted to use other tools.

Code locality to sequence graph (SG). When developing a program, structurally related functions are often placed close to each other in the source code by programmers or the programming paradigm. For example, the programming paradigm may place functions that operate on the same data next to each other such as the methods of a class. Source code locality transfers directly to the binary code because the compiler generates an object file for each source file and then the static linker concatenates the code (.text) sections of each object file to produce the code section of the final executable. Thus, functions that were next to each other in the source code (e.g., from the same source file or in the same class) end up being next to each other in the final executable. Code locality captures

the intuition that functions that are close to each other more likely belong to the same component. To generate the sequence graph, the functions identified by IDA are sorted in increasing order of their starting address. Then, directed edges are added between consecutive functions, directed from the function at lower address to the one at higher address.

Data references to data-reference graph (DRG). Functions operating on the same data are more likely to be structurally related, as they are related to the data semantics. This is especially true in object-oriented programming, where encapsulation makes data members be accessed by methods in the class. BCD constructs a data-reference graph by adding edges between functions that access the same variable. In an executable, global variables, static variables, and constant string literals are allocated statically with lifetime spanning across the entire program execution. In this work, a data reference is the offset of a statically allocated variable in the .data, .bss, or .rodata sections of an executable. For local variables and non-static class members, storage is dynamic. We focus on static data, global variables, and string literals because local variables in the stack do not reveal data references across functions. Moreover, data references to heap-allocated variables are difficult to analyze statically. To build the data-reference graph, BCD first creates a mapping between a function f and the set of statically allocated variables it references D. We denote this mapping as $\Phi : f \rightarrow D$, where $D = \langle D_1, \ldots, D_m \rangle$. Here, D_j is the offset of the jth static variable accessed by function f. An edge between two functions f^i and f^j is added when they reference at least one variable in common. To maintain the directed semantics of all graphs, an edge in each direction is added between the two functions. A larger set of common data references between two functions implies a stronger likelihood that both functions are part of the same component. Thus, an edge weight is assigned proportional to the number of common data references between the two functions. The weight is the same in both directions.

Function calls to call graph (CG). The final decomposition property that BCD leverages is that of function calling relationships. Intuitively, when function f^i calls function f^j, it is likely that those two functions are structurally related. If a set of functions call each other more than they call functions not in the set, then it is more likely that the set of functions belongs to the same component. BCD builds a call graph by adding a directed edge from function f^i to function f^j if f^i calls f^j. The larger the number of calls between two functions the stronger their structural relationship. This notion is captured by assigning an edge weight corresponding to the number of calls between two functions.

Challenges. In each of the above graphs, it is challenging to identify component boundaries—i.e., a set of edges that when removed results in a disconnected set of subgraphs, each representing a component. Particularly, boundaries between consecutive components (e.g., C++ classes) are unknown. This affects component boundary identification in a sequence graph. Similarly, component boundary identification in a data reference graph is affected by generic functions such as memcpy or printf that may operate on the same data, despite being largely unrelated to the set of related functions within

a component. Finally, it is not always true that a function calls another related function. For example, the `main` function may act as a dispatcher to other functions and is not contextually related to its callees. Thus, no single graph can be used to detect components. To address these challenges, the next step combines the three graphs into a decomposition graph.

2.2 Decomposition Graph Construction

While a graph built using a decomposition property contains information about structural relationships between functions, it may not contain sufficient information to identify components (i.e., disjoint subgraphs that represent groups of structurally related functions). To address this issue, our approach constructs a weighted and directed decomposition graph $H = (V', E', W)$, combining SG, CG, and DRG. Here, V' is the set of functions in the executable and E' is the union of all edges from the three decomposition properties. Graph edges are weighted according to the associated decomposition property. We denote an edge between function pair f^i and f^j by (f^i, f^j). For all $(f^i, f^j) \in E'$, BCD computes an edge weight w^{ij} as a linear combination of the corresponding edge weights in SG, CG and DRG. We assign a value of 1 to each edge weight in SG for indicating the relationship between consecutive functions.

Edge weight computation. The edge weights for each graph can be represented as an adjacency matrix M, in which each matrix element M_x^{ij} corresponds to an edge $(f^i, f^j) \in E_x$. Subscript $x \in \{s, d, c\}$ denotes SG, DRG, or CG, respectively. For nodes in SG, M_s consists of an adjacency matrix whose elements are each 1 or 0:

$$M_s^{ij} = \begin{cases} 1 & \text{if } (f^i, f^j) \in E_s \\ 0 & \text{otherwise} \end{cases}$$

However, in the case of DRG and CG, the corresponding matrix elements have a value equal to the count of common data references or function calls, respectively:

$$M_d^{ij} = \begin{cases} y_d & \text{if } (f^i, f^j) \in E_d, \text{, where } 0 < y_d \in \mathbb{N} \\ 0 & \text{otherwise} \end{cases}$$

$$M_c^{ij} = \begin{cases} y_c & \text{if } (f^i, f^j) \in E_c, \text{where } 0 < y_c \in \mathbb{N} \\ 0 & \text{otherwise} \end{cases}$$

where $\mathbb{N}$ denotes natural numbers.

While matrices for SG and CG directly capture the interaction strength of functions, M_d only captures the number of common data references. Particularly, the elements of M_d ignore the effect of dissimilarity in the globally ordered set of data references obtained from Φ. A typical linker orders data references according to the order in which functions refer to them. Therefore, it is more likely that two functions accessing far apart variables (according to the global order of data references) belong to different components compared to functions accessing nearby variables. We capture this notion using a dissimilarity score ρ_d between pairs of functions in E_d. For each $(f^i, f^j) \in E_d$, ordered lists of data references $D^i = \Phi(f^i)$ and $D^j = \Phi(f^j)$ are obtained from Φ for functions f^i and f^j, respectively. We use the *Levenshtein distance* [7], a popular string distance measure, to obtain the dissimilarity score between D^i and

D^j. Each element of the dissimilarity score matrix is given by

$$\rho_d^{ij} = \begin{cases} 1 - \frac{L(D^i, D^j)}{\max(p, q)} & \text{if } (f^i, f^j) \in E_d \wedge \max(p, q) > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $D^i = \Phi(f^i)$ has length p, $D^j = \Phi(f^j)$ has length q, and L denotes the Levenshtein distance. When the length of either D^i or D^j is 0, we assign $\rho_d^{ij} = 0$ since $L(D^i, D^j) = \max(p, q)$.

Given the three matrices (M_s, M_d, and M_c) and the dissimilarity matrix ρ_d, the combined edge weights are obtained using a linear combination of elements in matrices as follows. We first compute a penalty matrix N that computes the inverse distance between the ordered set of functions. Each element of N is given by

$$N^{ij} = \begin{cases} \frac{1}{|i-j|} & \text{if } i \neq j \\ 1 & \text{otherwise} \end{cases}$$

This penalty encourages the formation of components consisting of functions that are sequentially connected. Finally, the final edge weight matrix of the decomposition graph is given by

$$W = N \circ (\alpha M_s + \beta M_c + \gamma (\rho_d \circ M_d))$$

where α, β and γ are scalar hyperparameters, and operator $\circ$ denotes element-wise multiplication or Hadamard product. We empirically determine the value of each hyperparameter through cross-validation (see Section 3.1).

2.3 Partitioning

Our inductive assumption is that components are formed from disjoint sets of functions that primarily interact with other functions within the component, while interacting less with functions in other components. Since the number of components (or communities) in a given executable is unknown, we use Newman's generalized community detection algorithm [22], which does not require prior knowledge of the number of existing components (or communities). The algorithm has a time complexity of $O((m+n)n)$, where m is the number of edges and n the number of vertices in the graph. It optimizes a modularity function, where modularity Q is defined as the difference between the fraction of edges that fall within the given cluster and the expected fraction of edges if they were distributed at random. Modularity is widely used as a goodness measure for graph clustering, and is computed as $Q = \sum_i (e^{ii} - \sum_j e^{ij})$, where e^{ij} is the fraction of edges in the network that connect nodes in cluster C_i to those in cluster C_j (i.e., component boundaries).

The algorithm follows a bottom-up hierarchical clustering approach. It begins by considering each node as a separate community. It then merges the nodes connected with edges having optimal weights and detects the number of communities by optimizing the global modularity. Since our decomposition graph captures the function interactions as edge weights, we expect that related functions would be grouped inside the same component.

3 EVALUATION

In this section, we first present our experimental setup in Section 3.1 and then the evaluation results of BCD in Section 3.2.

			Source Code			Binary Code			
ID	**Program**	**Project**	**KLOC**	**Header Files**	**C++ Files**	**Type**	**Object Files**	**Classes**	**Methods**	**Funcs**
P1	DynamicDPI.exe	Dynamic DPI	4.6	19	17	PE	17	16	171	2,604
P2	genericucp.exe	UPnP Control Point	4.7	8	6	PE	6	7	68	634
P3	WRTPackageDebug.exe	WinRT Debug Tool	1.5	11	7	PE	7	7	38	488
P4	AmbientLightAware.exe	Ambient Light	1.6	6	5	PE	5	4	31	910
P5	BC6HBC7EncoderCS.exe	BC6HBC7 Encoder	1.8	5	5	PE	5	16	148	905
P6	CameraCapture.exe	CameraCapture UI	3.6	21	11	PE	10	8	144	3,226
P7	StarterKit.exe	VS 3D Starter Kit	7.0	18	10	PE	7	11	133	2,526
P8	AsyncDynamicObserver.exe	Multithreaded Login	2.5	14	11	PE	11	10	38	1,082
P9	DynamicObserver.exe	Multithreaded Login	1.5	11	8	PE	8	6	25	739
P10	DistributorMQ.exe	Multithreaded Login	2.5	14	11	PE	11	10	43	1,093
P11	DynamicShaderLinkage.exe	Dynamic Shader	44.2	20	16	PE	14	14	198	4,114
P12	7z.exe	7zip	27.2	41	69	PE	69	112	837	2,179
P13	7zG.exe	7zip	28.8	56	89	PE	89	149	890	2,530
P14	7zFM.exe	7zip	37.9	73	140	PE	140	182	979	3,149
P15	nping.exe	Nmap	23.4	21	18	PE	18	15	228	2,340
P16	nmap.exe	Nmap	58.3	50	48	PE	48	13	572	6,265
P17	cppcheck.exe	Static analysis tool	171.2	81	81	PE	78	78	668	2,248
P18	lzip	LZMA compressor	3.9	5	5	ELF	5	11	23	33
P19	tinyXMLTest	Tiny XML	7.1	2	5	ELF	5	14	138	2,744
P20	gperf	Gperf	8.3	14	11	ELF	11	19	81	58
P21	Astyle	Astyle	18.3	3	6	ELF	4	17	152	2,740
P22	re2c	Re2c	18.8	23	22	ELF	20	30	133	285
P23	lshw	Lshw	24.2	12	13	ELF	13	7	125	1,429
P24	smartctl	SMART disk analyzer	53.4	30	25	ELF	16	36	139	457
P25	pdftohtml	Pdf to html converter	91.4	87	87	ELF	85	125	1048	499

Table 1: Programs used in BCD **evaluation.**

3.1 Experiment Setup

We evaluate BCD using 25 programs. The dataset includes popular projects (e.g., Nmap, 7zip) collected from the Microsoft sample code repository [21], SourceForge [28], and the GNU software repository [12]. The dataset has executables from 4 projects having a size greater than 50 KLOC. These projects include multiple executables that often share code between themselves. In addition, 10 of the programs are benchmarks used for evaluating the Lego System [30], which recovers class hierarchies and composition-relationship using static and dynamic analysis techniques.

For all 25 programs, the source code is publicly available. Each program's source code is compiled with debugging information for extracting the ground truth needed to evaluate BCD. The ground truth is a mapping of methods in each class obtained from the debugging symbols, with corresponding class information. Since the source code may not be well structured or may not strictly follow modular programming principles, we extract the set of functions in each class and manually verify whether they form a component. Note that we only use the source code and debugging symbols to generate the necessary ground truth. BCD operates on the executables without access to source code or debugging symbols. For evaluating the effect of compiler optimization, we include PE (P1-P17) executables compiled using Visual Studio and ELF executables (P18-P25) compiled using g++.

Table 1 summarizes the 25 C++ programs. For each program, it shows the program identifier, the program name, the project the program belongs to, and source code and binary code statistics.

Evaluation metrics. We measure the overall performance of BCD by computing the Precision P, Recall R, and F_1 score values. The number of functions in a component may have high variance (i.e., a few components may have a small number of functions, while others may have a large number of functions). Such cases may mislead our analysis if a simple average of component scores is considered. Thus, we report a weighted average of scores [25] across all components. This *macro-averaged score* is computed as

$$P_w = \sum_{i=1}^{N_c} P_i \frac{n_i}{N_f}, \quad R_w = \sum_{i=1}^{N_c} R_i \frac{n_i}{N_f}, \quad F_1^w = \sum_{i=1}^{N_c} F_1^{(i)} \frac{n_i}{N_f} \quad (1)$$

where P_i, R_i, and $F_1^{(i)}$ are the scores for component C_i; N_f is the total number of functions in all components; n_i is the number of functions in component C_i; and N_c is the total number of components in the executable.

Graph weight hyperparameter training. In the decomposition graph construction, hyperparameters α, β, and γ determine the contribution of edge weights from the sequence graph, function-call graph, and data-reference graph, respectively. For normalization, we constrain $\alpha + \beta + \gamma = 1$. To empirically obtain the value of hyperparameters, we perform a grid-search over the range values. Specifically, given a dataset of binary executables, we apply 5-fold cross-validation by running BCD on the dataset for different values of $\alpha, \beta \in [0, 1]$ with a step size of 0.1, and assign $\gamma = 1 - \alpha - \beta$. The final choice of hyperparameter values corresponds to the maximum cross-validated average F_1^w score.
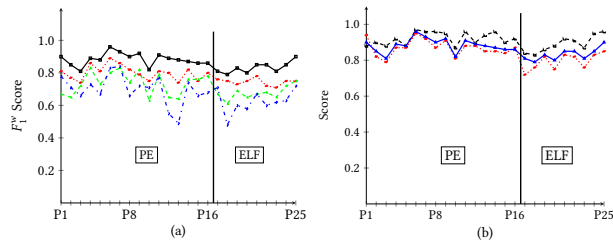
Figure 2: (a) F_1^w score comparison for decomposing programs using different feature sets. (‑ ◆ ‑ CG-BCD, ‑ ◀ ‑ DRG-BCD, ⋯◆⋯ DRG-CG-BCD, ━■━ All-BCD). (b) Weighted-macro-averaged ‑ ◆ ‑ precision, ⋯◆⋯ recall and ━▲━ F_1 scores.

Baseline. We compare the accuracy of BCD with three baseline methods that use some (but not all) of the decomposition properties. The first two baselines use a single decomposition property, i.e., either a CG (denoted by CG-BCD) or a DRG (denoted by DRG-BCD). We avoid using the SG as baseline since it would result in either a single component, or in each function forming its own component. The third baseline uses a combination of edges from DRG and CG to form the decomposition graph, denoted as DRG-CG-BCD. Finally, we denote BCD by All-BCD, emphasizing that edges from all three graphs are used to construct the decomposition graph.

3.2 Results

Figure 2-(a) shows the F_1^w scores for the decomposition into components output by BCD and the baseline methods. Here, *PX* indicates the program identifier as per Table 1. For both PE and ELF executables, the results clearly indicate that the highest F_1^w score is obtained by All-BCD, which outperforms the baseline methods. Moreover, the baseline method DRG-CG-BCD, which combines CG and DRG edges, performs better compared to the other baselines that use a single decomposition property (i.e. DRG-BCD and CG-BCD). This supports our claim that a linear combination of the edge weights from all three graphs improves the recovery of components compared to the individual decomposition properties. In addition, the difference in F_1^w between PE and ELF programs is negligible, indicating that BCD is robust to variations in executables produced by different compilers. Moreover, function re-ordering (from employing different compilers) can vastly affect the code locality property, i.e., the sequence graph. The results of DRG-CG-BCD perform close to All-BCD, indicating the robustness of BCD to such transformations by ignoring SG edges during partitioning.

Analysis of errors. As BCD automatically determines the number of components using Newman's community detection method, the resulting components may not always agree with the true number of components in an executable, according to the ground truth. A phenomenon called *under-splitting* occurs when functions truly belonging to two or more classes are grouped into a single component by BCD. Conversely, a phenomenon called *over-splitting* occurs when a set of functions truly belonging to a single class are distributed across multiple components.

We examine the performance of BCD in decomposing program executables into components using the weighted macro-averaged scores of precision, recall and F_1. As illustrated in Figure 2-(b), the median F_1^w score of programs containing PE executables (and corresponding variance) is 0.88±0.0036, and that of ELF executables is 0.84 ± 0.001. Particularly, we observed that component under-fitting occurs mainly in classes with only one or two functions. These small-sized components lack strong code and data locality features. For example, program P18 resulted in the least F_1^w score. This program has 7 classes, each having a single function. As a result, we observed 2 under-split partitions, each having functions from at least 3 different classes.

Hyperparameter sensitivity. We measure BCD's sensitivity to meta-weight parameter values by measuring the variance of α, β and γ obtained during the 5 fold cross-validation. The average value of meta-weight parameters yielding the highest F_1^w score in each run are $\alpha = 0.237 \pm 0.0026$, $\beta = 0.362 \pm 0.0026$, and $\gamma = 0.4 \pm 0.0028$. From these results, it can be observed that the performance of BCD is not significantly sensitive to the training data.

Runtime. We measure the runtime for BCD by aggregating the time spent during each step, i.e., decomposition properties extraction, decomposition graph construction, and partitioning. We use a Windows 32-bit machine with 4GB RAM and disassemble the executables using IDA [8]. The extraction of decomposition property graphs took 3s for the smallest program (P3) and 30s for the program (P24) with highest number of functions. Decomposition graph construction took an average of 10s per program. BCD spends a majority of time performing graph partitioning. Each iteration of the LinLogLayout toolkit [24] took a minimum time of 3s for P6, and a maximum time of 150s for P24.

4 RELATED WORK

There are numerous closely-related problems to binary decomposition. They include recovery of class hierarchy in C++ programs [30], code clone detection [27], program diffing [13, 34], and identification of functions with the same semantics [10, 15]. However, these differ from binary decomposition in their goal. Our goal is to statically decompose an executable into groups of structurally related functions. Functions belonging to the same component are likely to have related structural properties, but they may have very different syntactic representations and input-output relationships. For example, an encryption routine is highly related to its decryption routine and both are likely located in the same (cryptographic) component, but their input-output relationships are very different and can operate of very different data.

Most related are works on software module clustering [2, 19], which cluster program source code to recommend the best split into components to the developer. In contrast, our approach operates on program executables and tries to recover the component structure that the developer used, which are lost during compilation, rather than recommending a new component structure.

5 DISCUSSION

Obfuscation. When designing BCD, we assume that the binary is unobfuscated. In other words, our decomposition graph assumes

that the function sequence is unchanged. However, we also evaluated the robustness of BCD using only call-graph and data-reference graphs. In our test, when function sequence graph is excluded, BCD F_1^w score reduces to from 0.86 to 0.78 for C++ applications. Although an adversary could use obfuscation to defeat BCD, those obfuscations might well have the side-effect of raising detection alarms. For example, if BCD encounters a binary that seems to have extremely chaotic locality properties, that in itself could be used as a malware detection strategy. One way to address obfuscation is to de-obfuscate before applying BCD, e.g., using solutions such as dynamic unpackers [9].

Dynamic features. For simplicity, we have focused on features extracted statically from the executable. However, features extracted from program executions could also be incorporated into the decomposition graph, improving its efficiency. Some example dynamic features that may provide useful modularity information are functions used in a certain order and functions that access the same data structures in heap-allocated memory.

6 CONCLUSION

This paper introduced the problem of binary code decomposition and addressed its challenges by proposing a novel approach, called BCD, for decomposing a program executable into components. BCD takes a binary executable as input, and extracts code locality, data references, and calling relationships to build a decomposition graph. It then applies a graph-theoretic approach to partition the decomposition graph into disjoint components. Our evaluation results show that BCD is able to achieve a high precision and recall for decomposing the tested programs into components having structurally related functions.

REFERENCES

[1] Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. 2014. OBA2: An Onion Approach to Binary Code Authorship Attribution. In *Digital Investigation*, Vol. 11. S94–S103.

[2] Nicolas Anquetil and Timothy C. Lethbridge. 1999. Experiments with Clustering as a Software Remodularization Method. In *Proc. 6th Working Conf. Reverse Engineering (WCRE)*. 235–255.

[3] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proc. 23rd USENIX Security Sym*. 845–860.

[4] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-preserving Structural Analysis and Iterative Control-flow Structuring. In *Proc. 22nd USENIX Security Sym*.

[5] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. 2010. Binary Code Extraction and Interface Identification for Security Applications. In *Proc. 17th Annual Network & Distributed System Security Sym. (NDSS)*.

[6] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babic, and Dawn Song. 2010. Input Generation Via Decomposition and Re-Stitching: Finding Bugs in Malware. In *Proc. 17th ACM Conf. Computer and Communications Security (CCS)*. 413–425.

[7] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. 2004. Dictionary Matching and Indexing with Errors and Don't Cares. In *Proc. 36th Annual ACM Sym. Theory of Computing (STOC)*. 91–100.

[8] Chris Eagle. 2008. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA.

[9] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A Survey on Automated Dynamic Malware-analysis Techniques and Tools. *ACM Computing Surveys (CSUR)* 44, 2 (2012).

[10] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proc. 23rd USENIX Security Sym*.

[11] M. Van Emmerik and T. Waddington. 2004. Using a Decompiler for Real-world Source Recovery. In *Proc. 11th Working Conf. Reverse Engineering (WCRE)*. 27–36.

[12] Free Software Foundation. 1983. GNU Software Repository. www.gnu.org/software/software.html. (1983). Retrieved 3/30/2018.

[13] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *ICICS*.

[14] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller. 2011. Labeling Library Functions in Stripped Binaries. In *Proc. 10th ACM SIGPLAN-SIGSOFT Work. Program Analysis for Software Tools and Engineering (PASTE)*. 1–8.

[15] Lingxiao Jiang and Zhendong Su. 2009. Automatic Mining of Functionally Equivalent Code Fragments Via Random Testing. In *Proc. 18th Int. Sym. Software Testing and Analysis (ISSTA)*. 81–92.

[16] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. 2013. Rendezvous: A Search Engine for Binary Code. In *Proc. 10th Working Conf. Mining Software Repositories (MSR)*. 329–338.

[17] Dohyeong Kim, William N. Sumner, Xiangyu Zhang, Dongyan Xu, and Hira Agrawal. 2014. Reuse-oriented Reverse Engineering of Functional Components From x86 Binaries. In *Proc. 36th Int. Conf. Software Engineering (ICSE)*. 1128–1139.

[18] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. 2010. Inspector Gadget: Automated Extraction of Proprietary Gadgets From Malware Binaries. In *Proc. 31st IEEE Sym. Security & Privacy (S&P)*.

[19] Spiros Mancoridis, Brian S. Mitchell, Yihfarn Chen, and Emden R. Gansner. 1999. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proc. IEEE Int. Conf. Software Maintenance (ICSM)*. 50–59.

[20] Xiaozhu Meng. 2016. Fine-grained Binary Code Authorship Identification. In *Proc. 24th ACM SIGSOFT Int. Sym. Foundations Software Engineering (FSE)*. 1097–1099.

[21] Microsoft. 2007. Visual Studio sample codes. code.msdn.microsoft.com/vstudio. (2007). Retrieved 3/30/2018.

[22] Mark E.J. Newman. 2004. Fast Algorithm for Detecting Community Structure in Networks. *Physical Review E* 69, 6 (2004), 066133.

[23] Beng Heng Ng and Atul Prakash. 2013. Exposé: Discovering Potential Binary Code Re-use. In *Proc. 37th IEEE Annual Computer Software and Applications Conf. (COMPSAC)*. 492–501.

[24] Andreas Noack. 2009. LinLogLayout: Graph Clustering and Force-directed Graph Layout. code.google.com/p/linloglayout. (2009). Retrieved 3/30/2018.

[25] Arzucan Özgür, Levent Özgür, and Tunga Güngör. 2005. Text Categorization with Class-based and Corpus-based Keyword Selection. In *Proc. 20th Int. Sym. Computer and Information Sciences (ISCIS)*. 606–615.

[26] Nathan Rosenblum, Xiaojin Zhu, and Barton P. Miller. 2011. Who Wrote This Code? Identifying the Authors of Program Binaries. In *Proc. 16th European Conf. Research in Computer Security (ESORICS)*. 172–189.

[27] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting Code Clones in Binary Executables. In *Proc. 18th Int. Sym. Software Testing and Analysis (ISSTA)*. 117–128.

[28] Slashdot Media. 1999. SourceForge. sourceforge.net. (1999). Retrieved 3/30/2018.

[29] Randy Smith and Susan Horwitz. 2009. Detecting and Measuring Similarity in Code Clones. In *Proc. 3rd REF/TCSE Int. Work. Software Clones (IWSC)*. 28–34.

[30] Venkatesh Srinivasan and Thomas Reps. 2014. Recovery of Class Hierarchies and Composition Relationships From Machine Code. In *Proc. 23rd Int. Conf. Compiler Construction (CC)*. 61–84.

[31] Wenhao Wang, Xiaoyang Xu, and Kevin W. Hamlen. 2017. Object Flow Integrity. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. 1909–1924.

[32] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. Behavior Based Software Theft Detection. In *Proc. 16th ACM Conf. Computer and Communications Security (CCS)*. 280–290.

[33] Fangfang Zhang, Yoon-Chan Jhi, Dinghao Wu, Peng Liu, and Sencun Zhu. 2012. A First Step Towards Algorithm Plagiarism Detection. In *Proc. 21st Int. Sym. Software Testing and Analysis (ISSTA)*. 111–121.

[34] Zynamics. 2004. BinDiff. www.zynamics.com/bindiff.html. (2004). Retrieved 3/30/2018.