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@ece 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

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 \( F_x \) with \( x \in [1, m] \). These functions can be associated with \( k \) modules or components, denoted \( C_x \) 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,
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 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).
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 \( M_{ij} \) corresponds to an edge \((f^i, f^j) \in E\). Subscript \( \times \in \{s, d, c\} \) denotes SG, DRG, or CG, respectively. For nodes in SG, \( M_s \) consists of an adjacency matrix whose elements are each 1 or 0:

\[
M_{ij}^s = \begin{cases} 
1 & \text{if } (f^i, f^j) \in E_s \\
0 & \text{otherwise}
\end{cases}
\]

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:

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

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

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 \( \Phi \). 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 \( \rho_d \) 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 \( \Phi \) 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

\[
\rho_{ij}^d = \begin{cases} 
1 - \frac{L(D^i, D^j)}{\max(p, q)} & \text{if } (f^i, f^j) \in E_d \land \max(p, q) > 0 \\
0 & \text{otherwise}
\end{cases}
\]

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 \( \rho_{ij}^d = 0 \) since \( L(D^i, D^j) = \max(p, q) \).

Given the three matrices \((M_s, M_d, M_c)\) and the dissimilarity matrix \( \rho_d \), 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

\[
W = N (aM_s + bM_c + \gamma (\rho_d \odot M_d))
\]

where \( a, b \) and \( \gamma \) are scalar hyperparameters, and operator \( \odot \) 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

\[
Q = \sum_{C_j} \left( e^{ij} - \frac{1}{n} \right),
\]

where \( e^{ij} \) is the fraction of edges in the network that connect nodes in cluster \( C_j \) to those in cluster \( C_i \) (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.
We evaluate BCD 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

$$P_w = \frac{1}{N_C} \sum_{i=1}^{N_C} P_i n_{iC}$$

$$R_w = \frac{1}{N_C} \sum_{i=1}^{N_C} R_i n_{iC}$$

$$F_1 = \frac{P_i + R_i}{2}$$

where $P_i$, $R_i$, and $F_1$ are the scores for component $C_i$; $N_C$ 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 $\alpha$, $\beta$, and $\gamma$ 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.
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 $0.88 \pm 0.0036$, and that of ELF executables is $0.84 \pm 0.001$. 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 $\alpha$, $\beta$ and $\gamma$ 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 $F^w$ 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 execution 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.

### ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable comments. This work was partially supported by AFOSR awards FA9550-14-1-0119 and FA9550-14-1-0173, ONR awards N00014-14-1-0030 and N00014-17-1-2995, and NSF awards 1054629 and 1513704. Partial support was also provided by the Regional Government of Madrid through the N-GREENS Software-CM project S2013/ICE-2731, the Spanish Government through the DEDETIS grant TIN2015-7013-R, and the European Union through the ElasTest project ICT-10-2016-731535.

### REFERENCES


