SANDIA REPORT
SAND2020-9494
Printed September 10, 2020

Sandia National Laboratories

Using modularity to segment binary code

Jacek Skryzalin, Daniel Chivers

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ABSTRACT

We consider the problem of recovering program structure from compiled binary code. We first extract the call graph and layout of functions in memory from the compiled code and represent this information in a graphical format. We then employ Louvain's modularity algorithm to identify clusters of functions that are considered to be related. We find that the quality and properties of clusters extracted by our technique are greatly impacted by the relative importance we assign to the call graph and the ordering of functions in memory.

ACKNOWLEDGMENT

I'd like to thank Kathy Simonson for providing the opportunity to work on this project and for many helpful discussions throughout the project, and to the reviewers who provided many helpful comments that greatly improved this report.

Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

DWARF a debugging data format (potential backronym: Debugging With Attributed Record Formats)

ELF Executable and Linkable Format (formerly Extensible Linking Format) – a file format used for executable files and other binary code

HTML HyperText Markup Language – a markup language used to display documents in a web browser

1. RECOVERING PROGRAM STRUCTURE FROM COMPILED BINARY CODE

1.1. Introduction

Software engineering is the art of applying fundamental engineering principles when designing large, scalable software. The field of software engineering spans a wide breadth of activities; these include requirements gathering, design, development, testing, and maintenance. Software design is the process during which one creates a blueprint of the software system, taking into consideration the requirements of the system. For larger projects, one major goal of software design is the management of the complexity of the program. This is largely done through the principles of abstraction, encapsulation, modularization, and hierarchy [9]. Here, modularization refers to the practice of dividing the program into relatively self-contained components, or *modules*. Typically, each module encapsulates a fundamental set of related functionalities. For example, a web browser could have modules for the user interface, the network interface, and rendering HTML and other code to the screen. Each module may additionally be divided into a number of submodules. For example, the network interface module may have a submodule that implements security features for the browser.

A well-designed program that exhibits a high degree of modularization will be easier to develop (e.g., because different modules can be developed independently by different programmers) and maintain (e.g., because someone who wishes to find and fix a bug will have a better understanding of the overall structure of the program and will be better able to isolate the bug). Throughout the development process, modularization is implemented by placing the source code for different modules in different files or directories.

During the compilation process, however, these human-understandable artifacts of modularization are lost, as the compiler will combine and merge all functions into one file. Thus, when reverse engineering compiled code (e.g., for a security audit), an analyst is faced with a large collection of functions; most of these functions will not be relevant to the specific question the analyst wishes to answer. Understanding where to look within the binary file to answer key questions can take a large amount of analyst time. It is thus the goal of this report to describe a method by which some of the modules of the program may be isolated and recovered automatically.

Our approach is to perform a cluster analysis on the set of functions so that functions defined in proximity to one another and functions that frequently call one another will belong to the same cluster. In particular, we augment the function call graph of the program with edges that reflect proximity in address space and employ Louvain's modularity community detection algorithm to create clusters of functions. By changing the relative importance of the call graph and address space proximity, we can capture and recover different notions of modules.

Our approach relies on two major assumptions. First, we assume that modularization is reflected in both the function call graph and the directory structure. That is, we assume that functions defined within one directory call one another much more than they call functions in other directories. This assumption allows us to extract program modules by identifying tightly knit clusters in the function call graph. Moreover, this assumption allows us to use directory structure as a proxy for ground truth.

Next, we assume that the compiler preserves function locality. That is, we assume that the compiler will place functions defined in the same file at similar locations in address space, and that given two files in the same directory, the compiler will place the two sets of functions defined in these files close together in address space. Although this assumption holds for the most frequently used compilers, there are notable exceptions (e.g., [16]) that employ "binary scrambling" and other techniques as countermeasures against cyber-attacks. Although techniques like binary scrambling can prevent some automated analyses, many of these techniques do not serve as a major hindrance to a reverse engineer.

We also assume that we can discern from the compiled binary program which functions call one another without running the program (e.g., using static analysis). This assumption is mostly satisfied for programs compiled from most non-object-oriented languages (e.g., C). When code has been compiled from an object-oriented language (e.g., C++), the decision of which function to call can be made at runtime (e.g., through a virtual table) and is difficult to determine statically. Although we expect that the methods discussed here could be extended to take into account more exotic control structures like virtual tables, we assume that these are not present in the code that we examine.

We test our approach on six different compilations of the Linux kernel compiled with different compilers (clang and gcc) and with various levels of optimization (-Os, -O2, and -O3). There is not an appreciable variation in the homogeneity of the resulting clusters across different compilations of the kernel. This is somewhat expected – it is reasonable to assume that the majority of changes to compiled code that are observed when varying the compiler and optimization flags will affect the code produced within each function while generally leaving unchanged the call graph and the general order in which functions are placed in memory[1]. Although not perfect, our technique is largely able to create clusters consisting of similar functions.

The report proceeds as follows. In Section 1.2, we discuss previous work related to program analysis and graph community detection. In Section 1.3, we present our proposed technique – using Ghidra to extract information about a binary executable file, creating graphs that encode function similarity using the information extracted by Ghidra, and performing Louvain community detection to cluster functions. Section 1.4 contains the results of applying our technique to the Linux kernel. Section 1.4.1 discusses how we extract information about the Linux kernel, Section 1.4.2 discusses the extent to which our assumptions hold in the Linux kernel, and Section 1.4.3 provides a quantitative evaluation of our proposed technique using the homogeneity score. In Section 1.4.4, we provide a more qualitative evaluation by analyzing how

[1] Notable exceptions to this assumption include optimization flags like gcc's "`-finline-functions`" flag that encourages functions to be integrated into their callers.

well our technique extracts the set of functions related to function tracing in the Linux kernel. Finally, we conclude our report with a discussion of our findings in Section 1.5.

1.2. Related work

1.2.1. Program analysis and program graphs

Most work performed in computer program analysis focuses on understanding code at a micro-level. Tools like Ghidra [1], Radare [2], and IDA [18] allow an analyst to step through and reverse engineer smaller sections of code. These tools are incredibly useful when an analyst has a specific question in mind and will be focused on a relatively small and easily identified portion of the binary file. On the other hand, when attempting to gain a general understanding of a binary file, one might use Linux utilities like `file` and `readelf` to learn information about file types, program and section headers, and defined symbols. Within an executable binary file, the `.text` section contains the actual code executed. Unfortunately, there are currently very few analytics for acquiring a general understanding of this `.text` section. It is our goal to introduce one such analytic.

Within the realm of program analysis, researchers often make use of graphical representations of programs. There are three commonly used graphical representations of programs – the control flow graph, the data flow graph, and the call graph [15]. The control flow graph collapses into one node any sequence of machine instructions that are always executed together (e.g., sequences without any branches or jumps); edges in the control flow graph represent potential transitions between these sequences. The data flow graph represents each instance of data as one node and dependencies between various data values as edges. Whereas the control flow and data flow graphs are most commonly used to understand the inner workings of a function, the call graph represents how different functions of a program call one another [14]. These graphs can be used by an analyst looking at either source code or compiled binary code to gain a better understanding of the program. There are a number of analytics that make use of these graphs. For example, one can use control flow graphs indicative of malware to detect malicious code [11, 12], one can use dynamic traces of a running program in combination with the call graph to detect non-crashing bugs in code [14], and one can interpret anomalous subgraphs as indications that a bug may exist [21].

Analysts often look at only small portions of these graphs – an analyst will identify a variable or a sequence of instructions of interest and use the data flow or control flow graph to understand how that variable was derived or how that sequence of instructions might be executed. It is less common for analysts to apply global analytics to these graphs to gain a broader understanding of the program as a whole. We thus propose to apply global graph analytics to graphs derived from binary code in order to better understand the general purpose of the program.

1.2.2. Automated techniques for program analysis

Researchers have long performed analytics on source code [13, 27]. The semantic information provided in source code, which can include variable names, function names, data types, and data structures, is often much more rich and complete than that in compiled binary code. Furthermore, source code abstracts away details specific to the implementation of a computer program on hardware; what remains is a representation of a program that focuses more on *what* a program does rather than *how* the program is executed on computer hardware. Recently, there has been a flurry of work on source code that uses neural representations to perform downstream tasks (e.g., variable or method name prediction, identification of similar functions, bug detection) [4, 6, 17, 19, 22, 28, 30]. Much of this research takes into account the control flow and data flow information that can be derived from source code. Some research further considers innovative methods of constructing data used to train an algorithm; for example, Henkel et al. generate training data from a program using symbolic execution [17].

Some of the techniques used to analyze source code can port to binary code; others cannot. Indeed, any techniques that rely on semantic information in source code will not work with binary code. As a result, the analytics that one can immediately run on binary code are somewhat limited. Most research concerning analytics applied to binary code falls into two categories. In the first category, researchers attempt to fingerprint binary code to identify the compiler or author of the code [7]. In the second category, researchers attempt to identify similar functions [20, 26, 29].

Our approach differs from most published research in two respects. First, our analysis is more global; we aim to help uncover the general structure of an entire program. Second, our hope is that our analytic might be used by a security analyst to understand a binary program. In this respect, our goals are similar to those of Rosetta [3], which aims to assist analysts in recovering data structures from binary code.

1.2.3. Graph community detection

Graph community detection is the process of finding groups of nodes in graphs (called *communities*) such that the density of edges within each community is higher than that between communities. In this work, we use the Louvain community detection algorithm [8], which attempts to maximize the *modularity* Q of the community assignments[2]. Modularity is defined mathematically as

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \gamma \frac{k_i k_j}{2m} \right] \delta(c_i, c_j),$$

where $A_{i,j}$ represents the weight of the edge between nodes i and j, $k_i = \sum_j A_{i,j}$ is the sum of the weights of the edges attached to node i, $m = \frac{1}{2} \sum_{i,j} A_{i,j}$ is the sum of all edge weights, and $\delta(c_i, c_j)$ is equal to 1 if the community assignments c_i and c_j of node i and node j are identical and 0 otherwise. The parameter $\gamma > 0$ controls the resolution of the communities [25]; higher values of γ tend to yield community assignments with more communities. Although the problem of finding

[2] We use "modularity" to refer to this mathematical quantity Q and "modularization" to refer to the practice of organizing large programs into modules.

the community assignments that exactly maximize this quantity is NP-complete [10], there are a number of algorithms which approximately maximize modularity. The Louvain algorithm, one such algorithm, has been the de facto standard for the past decade. The algorithm initializes a separate community for each node and iteratively combines communities if doing so increases modularity. The algorithm terminates when no more of these combinations are possible. As the Louvain algorithm is randomized, the results of running the Louvain algorithm multiple times on the same graph are expected to differ. Nonetheless, the algorithm is relatively stable on most real-world data.

1.3. Methods

Our overall approach is to construct graphs that encode properties that facilitate separation of modules, combine these graphs, and perform Louvain community detection on the combined graph. We perform experiments with both directed and undirected graphs, and with both weighted and unweighted graphs. Our default is to use weighted, directed graphs. In our experiments, we did not observe an appreciable change in performance when moving from weighted to unweighted graphs, or when moving from directed to undirected graphs[3]. To create an undirected graph from a directed graph, we first collapse and combine any edges from n to n' and from n' to n, summing edge weights as necessary. To create an unweighted graph from a weighted graph, we replace all edge weights with weight 1.

We first reconstruct a call graph from function information extracted from the compiled binary. The nodes of this graph represent functions, and an edge from node n to node n' signifies that the function f corresponding to node n calls the function f' corresponding to node n'. Furthermore, the weight associated to the edge from n to n' is the number of `call` instructions within the code of f with a target of f'.

We next construct an address similarity graph, assuming that a function's instructions are consecutive and that instructions and addresses belong to only one function. We first construct a sorted list of the start addresses of all functions. We then construct a graph where an edge between two nodes indicates that two functions are defined near each other. Given functions f and f' defined at addresses a and a' and represented by nodes n and n', we draw an edge from n to n' if the distance d between a and a' in the sorted list of function addresses is less than δ (we use $\delta = 4$ as default in our experiments)[4]. Note that, if our graph is directed, there will be an edge from n to n' if and only if there is an edge from n' to n. Furthermore, if the graph is weighted, we assign a weight $g(d)$ to the edge. We experiment with the following options for g:

- $g(d) = \frac{1.0}{(1.0+d)^{\alpha}}$
- $g(d) = \alpha - d$

[3]The lack of a noticeable change in performance when moving from a weighted graph to an unweighted graph may be partially explained by the fact that, given any two functions f and f', it was unusual in the code we looked at for f to have more than one or two `call` instructions to f'. The lack of a noticeable change in performance when moving from a directed graph to an undirected graph may be partially attributed to the fact that, given two functions f and f', we rarely saw situations where both f calls f' and f' calls f.

[4]The distance δ refers to the index of a function in a list, not to the distance between functions in address space.

- $g(d) = e^{-\alpha|d|}$
- $g(d) = e^{-\alpha d^2}$

Because performance is relatively similar for most choices of g and most "reasonable" choices of α, we use the function $g(d) = \frac{1.0}{(1.0+d)^{0.5}}$ to weigh address similarity in all experiments discussed in Section 1.4.

We then combine the call graph and address similarity graph[5]. We construct the graphs so that the edge weights of the address similarity graph together contribute λ_{addr} times the total edge weights of the call graph[6]. For directed graphs, we first divide each edge weight in each component graph by the sum of all weights in that component graph; for undirected graphs, we divide each edge weight in each component graph by *twice* the sum of all weights in that component graph. We then multiply the edge weights in the address similarity graph by λ_{addr}. We then construct a new graph whose edge weights are equal to the sum of the (scaled) edge weights of the component graphs.

Finally, we use Louvain's modularity-based community detection algorithm to obtain n clusters[7]. We try to obtain $n = 100$ clusters; for the Linux kernel, we found that $n = 100$ allows for the formation of homogeneous clusters of functions without providing an overwhelming number of clusters. Although the Louvain algorithm does not allow the number of clusters to be set, we can alter the resolution parameter γ to achieve a desired number of clusters. In particular, we set bounds for γ and perform a binary search within these bounds until we find a value for γ that produces the desired number of clusters. We perform a maximum of iter $\in [10, 30]$ iterations of the search, each with a different value of γ (the specific value of iter used depends on the experiment; if the desired number of clusters is not seen by the iter$^{\text{th}}$ value of γ, we use the clustering obtained during the final run of the Louvain algorithm. As the Louvain algorithm is randomized, the number of clusters produced for a given value of γ is also a random quantity; in our experiments, we were almost always able to find a clustering of the functions such that the number of clusters was within 5% of the desired number of clusters.

1.4. Experiments

1.4.1. Data collection

We compile six different versions of the Linux kernel – we use two different compilers (clang and gcc), each with three levels of optimization (-Os, -O2, and -O3). In order to obtain ground truth from the binary files, we compile Linux with the debug information; as such, the produced binaries are substantially larger than fully stripped versions of the Linux kernel. Debug

[5] Because the nodes are identical in the two graphs, combining the graphs amounts to no more than altering edge weights.

[6] Note that modularity only considers the *relative* weights of the edges; the algorithm is unaffected by scaling all weights by some constant factor.

[7] Although it is common to call the groupings of nodes "communities" in graph analytics, we call our groupings "clusters" here.

Compiler	Optimization	Size of binary	*n* fcns	*n* conn. fcns	*n* disconn. fcns
gcc	-Os	648 MB	56,826	49,021	7,805
gcc	-O2	678 MB	50,542	42,396	8,146
gcc	-O3	717 MB	45,988	38,811	7,177
clang	-Os	530 MB	50,784	42,521	8,263
clang	-O2	556 MB	47,023	39,382	7,641
clang	-O3	560 MB	46,843	39,267	7,576

Table 1-1. Information about the compiled binaries of the Linux kernel. It should be noted that there are a surprising number of functions that did not directly call nor were called directly by another function. The number of such functions in each binary is given in the rightmost column of the table.

information is required only to determine ground truth[8]; we claim that our analytics will apply equally as well to fully stripped binaries. A summary of the produced binaries can be found in Table 1-1. Unless otherwise noted, our experiments are performed on the version of the Linux compiler compiled with gcc using -O2 optimization.

Ground truth is given by a map of functions to source file paths (e.g., my_function may be mapped to /path/to/source_file.c). Note that functions are uniquely identified by address (offset in the binary). To produce ground truth data, we first extract DWARF (ELF debug) information using `objdump -dwarf=decodedline`[9]. The output contains mappings from offsets in the binary to the source file path and the line number where the function is defined. We then use Ghidra [1] to iterate through all functions that Ghidra detects. We can be relatively certain that the functions that Ghidra has detected are exactly the functions present in the binary file due to the debug information present in our binary files. Given the function address, we use the output of `objdump` to look up the source file[10].

After we obtain ground truth for each function, we again use Ghidra to produce more detailed information about each function. A summary of the extracted data is given in Table 1-2. Note that not all of this information is used by our analytics.

1.4.2. Validating our assumptions

As discussed previously, our approach relies on assumptions about compiled code. Namely, we assume that functions defined within the same directory tend to call one another much more than

[8] We also use debug information to inform and expedite the process of extracting functions and the call graph from the binary file. Recovering functions and the call graph is possible without debug information but may require more effort.

[9] Another option is to use ctags to extract the function-to-source mapping. This does not work well because functions can share the same name across different source files. Additionally, the build process on a given source tree often results in more than one binary file being produced (e.g., a main program and several utilities). It would be difficult to align functions and source files to a specific binary because the build process may differ for different source trees.

[10] Ghidra has support for DWARF, but it is not easy to obtain the source location information directly from Ghidra; `objdump` is a good intermediate step.

Item	Notes
Address	
Name	Used for reference only; not used for graph analytics
ID	Unique Ghidra ID number
Stack frame size	Size of local variables
Number of arguments	
Return type	Accurate only when debug information is present
Size	Total function size in bytes
Types of arguments	Accurate only when debug information is present
Global and read-only data references	Address of each global or read-only data referenced by the function
Functions called	A list of functions called, along with the number of times each function is called; only direct calls are used at this time due to the difficulty in recovering the targets of indirect calls

Table 1-2. Information about each function that was extracted using Ghidra.

they call functions defined in another directory, and we assume that the compiler preserves function locality. In this section, we calculate a set of statistics that reveal the degree to which these assumptions are valid. Occasionally, functions cannot be mapped back to a source file (e.g., statically linked library functions). We lump all such functions into their own directory for this analysis. When reading these results, it will be helpful to keep in mind that the version of Linux we compile has 231 directories, 50,542 functions total, and that we are trying to group these functions into 100 clusters. Furthermore, there are only 176 directories which define over 20 functions; these directories account for 50,092 out of 50,542 functions defined.

Figure 1-1 shows the distribution of the number of functions defined per directory. This figure has an extremely long tail; indeed, 131 of the 231 directories define fewer than 100 functions, and over half of all functions are defined in only 23 directories.

To evaluate our assumption that the compiler preserves function locality, we examine the extent to which functions defined in the same directory are placed next to one another in address space. We find that functions are generally placed next to one another in a number of contiguous regions, which we call *address ranges*. Figure 1-2 shows that the vast majority of directories define their functions in fewer than 5 such contiguous regions. Generally, the directories defining functions that are dispersed in address space are either “include” directories or directories containing architecture-specific code. Figure 1-3 shows the distribution of the fraction of functions in each directory that are defined in the largest address range. We see that, generally, the majority of functions are defined in the largest address range. Indeed, at least 50% of functions are defined in the largest address range in 74% of the directories. When we restrict to the directories defining at least 20 functions, this statistic drops slightly; at least 50% of functions are defined in the largest address range in 69% of such directories. We conclude that although our function locality assumption is mostly valid, we must recognize that there are a number of exceptions to this

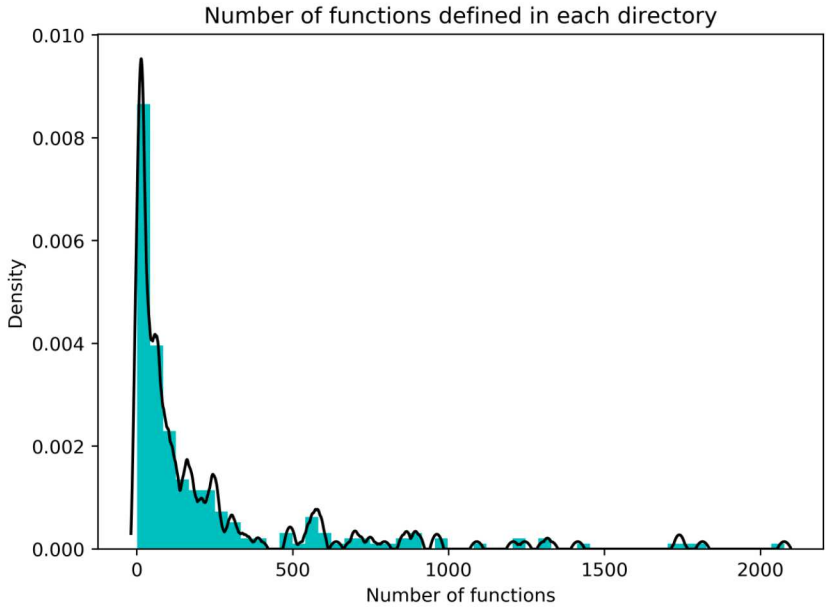

Figure 1-1. Kernel density estimate showing the distribution of the number of functions defined per directory in Linux kernel source code.

rule.

To evaluate our assumption that functions will tend to call other functions within the same directory, we look at the (weighted) fraction of function calls that occur within a directory. We weigh the function call from the function f to the function g_i by $\frac{1}{\mathrm{ct}_f}$, where $\{g_i\}_{i=1}^{\mathrm{ct}_f}$ is the set of size ct_f of functions called by the function f. The number of calls from f to g_i is not taken into account in this analysis[11]. For each directory that defines at least 10 functions, we calculate three (weighted) fractions:

- [incoming calls]: the (weighted) fraction of all calls whose target is in the directory such that the caller is also in the directory,
- [outgoing calls]: the (weighted) fraction of all calls made by functions of the directory whose target is also in the directory, and
- [all calls]: the (weighted) fraction of all calls whose caller or callee belongs to the directory that do not involve a function outside the directory.

The distribution of such fractions can be seen in Figure 1-4. Generally, the plurality (although not the majority) of function calls occur within a directory, regardless of whether we are looking at incoming calls, outgoing calls, or all calls. Interestingly, we see that, when choosing uniformly at random a directory d, a function call c into d, and a function call c' made by a function defined in d, the probability that c was made by a function defined in d is much higher than the probability

[11] Because most function calls occur with low multiplicity, we obtain very similar results when the call multiplicity is taken into account.

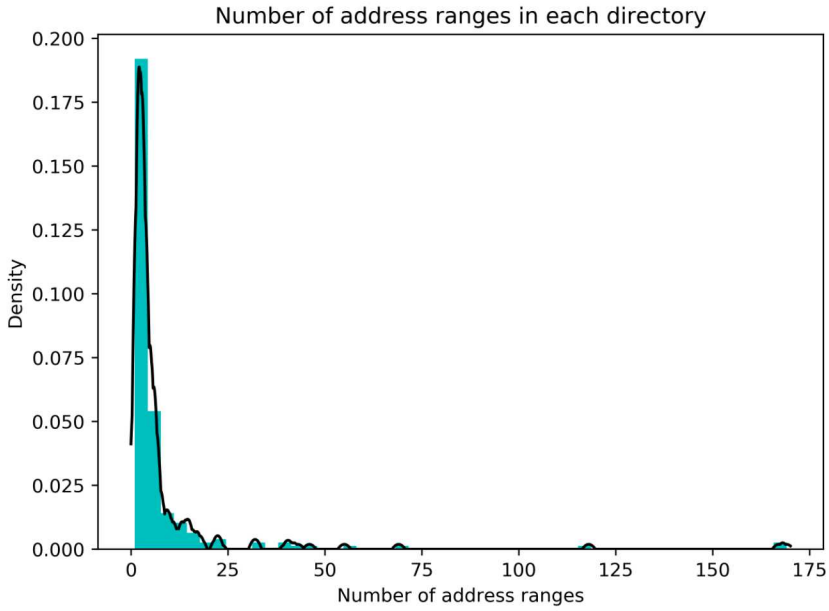

Figure 1-2. Kernel density estimate showing the distribution of the number of address ranges per directory in Linux kernel source code.

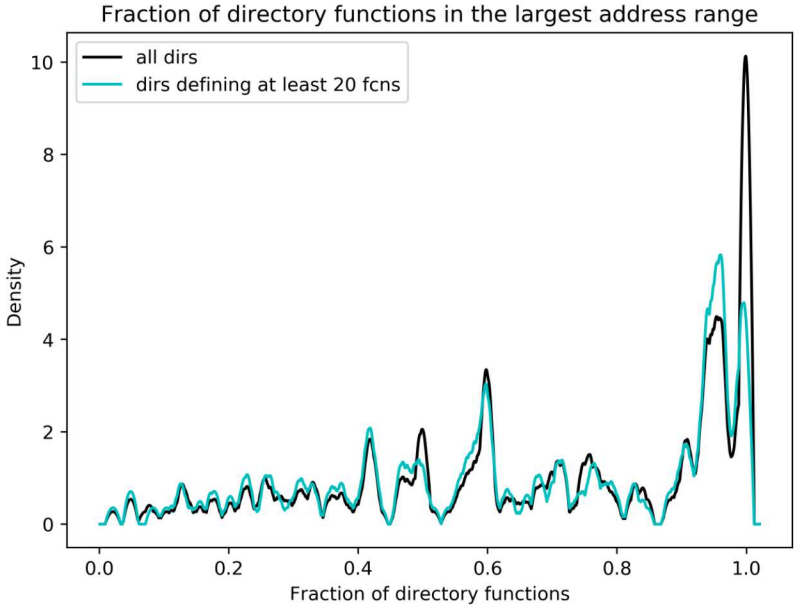

Figure 1-3. Kernel density estimate showing the distribution of the fraction of functions in each directory that are defined in the largest address range in Linux kernel source code.

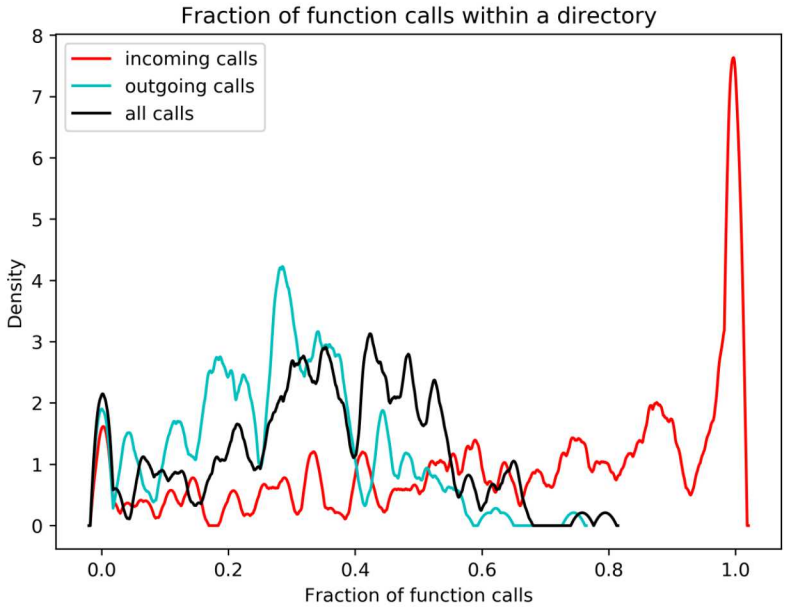

Figure 1-4. Kernel density estimates showing the distribution of the (weighted) fraction of calls occurring within a directory. We construct separate estimates for incoming function calls, outgoing function calls, and all function calls. We consider only directories defining at least 10 functions.

that the target of c' is also in d. Said differently, on average, function calls are much more likely to originate in the same directory than they are to have a target in the same directory.

1.4.3. Analytical results

Below, we discuss the results of our numerical experiments. We report the goodness of a function clustering using *homogeneity* [24]. The homogeneity of a clustering is an entropy-based measure of the similarity of items within each cluster. The homogeneity is a number in the closed interval $[0,1]$, with larger values indicating a clustering which more closely matches ground truth. We assume a set $K = \{k_1, ..., k_m\}$ of clusters, classes $C = \{c_1, .., c_n\}$, and a contingency table $\{a_{i,j}\}_{i,j}$, where $a_{i,j}$ is the number of items belonging to class c_i in cluster k_j. When all data belongs to the same class, homogeneity is defined to be 1. Otherwise, homogeneity is defined as

$$1 - \frac{H(C|K)}{H(C)},$$

where $H(C)$ (resp. $H(C|K)$) denotes the entropy of the class assignments (resp. the conditional entropy of the class assignments given cluster assignments):

$$H(C) = -\sum_{i=1}^{|C|} \frac{|c_i|}{N} \log \frac{|c_i|}{N}$$

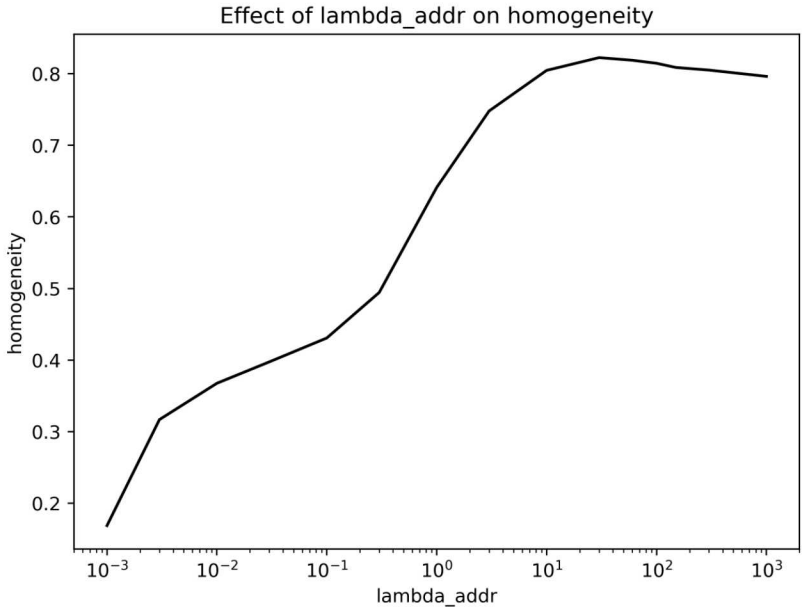

Figure 1-5. The effect of λ_{addr} on homogeneity.

$$H(C|K) = -\sum_{i=1}^{|C|}\sum_{j=1}^{|K|}\frac{a_{i,j}}{N}\log\frac{a_{i,j}}{|k_j|}$$

When calculating the homogeneity score, we ignore any functions that cannot be mapped back to a source file.

As mentioned previously, of all possible parameters, λ_{addr} has the largest effect on homogeneity. Indeed, the effects of all other parameters are hardly noticed, provided one chooses "reasonable" parameter values (e.g., those relatively close to the defaults discussed above). Here, we examine the effect of altering λ_{addr} on the homogeneity score. In particular, we calculate the homogeneity score for values of λ_{addr} ranging from 0.001 to 1000. For each such value, we calculate the average homogeneity score of performing 10 clusterings. The results of this experiment are visualized in Figure 1-5. We see that homogeneity is maximized when $10 \leq \lambda_{addr} \leq 100$. Because the highest homogeneity score is not seen at the lowest or highest considered values of λ_{addr}, Figure 1-5 also suggests that we can obtain a higher homogeneity when combining information from both the function layout and the call graph than when using information from one of those alone.

The parameter λ_{addr} also has a somewhat large effect on the distribution of the sizes of the clusters produced by our technique. Figure 1-6 shows three kernel density estimates of the cluster sizes produced with varying values of λ_{addr}. We see that when λ_{addr} is large, the distribution of cluster sizes is roughly Gaussian. However, as λ_{addr} decreases, the distribution becomes skewed to the right – there are a few larger clusters and many smaller clusters. When $\lambda_{addr} = 0.05$, there is a large percentage of clusters whose size is very small; we keep the bandwidth of the kernel density estimate small to better show this phenomenon. Table 1-3 shows the percentage of clusters whose

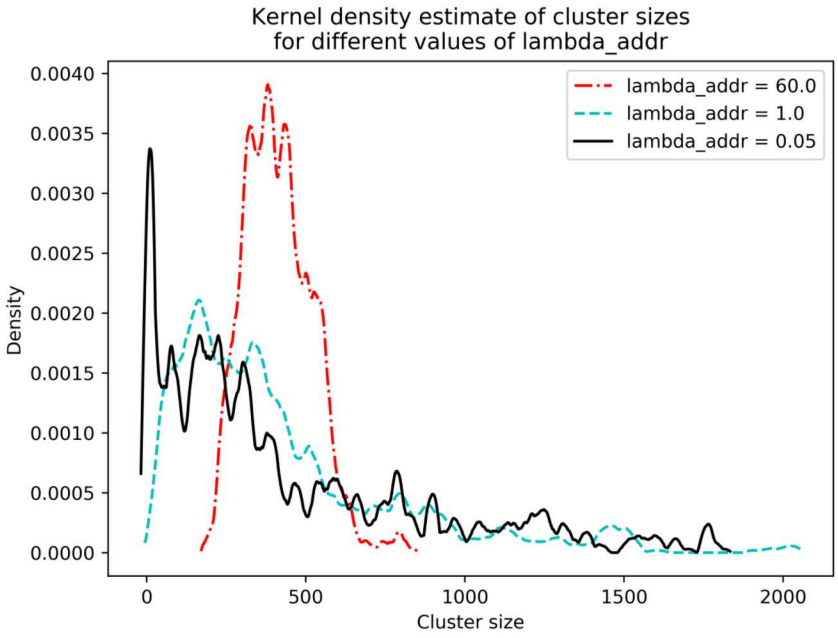

Figure 1-6. Kernel density estimates showing the distributions of cluster sizes produced while varying λ_{addr}.

Cluster size	1 - 9	1 - 49	1-99	100-749	750+	1000+	2000+
$\lambda_{addr} = 60.0$	0.0%	0.0%	0.0%	99.1%	0.9%	0.0%	0.0%
$\lambda_{addr} = 1.0$	0.0%	3.2%	11.1%	72.6%	16.3%	8.0%	1.5%
$\lambda_{addr} = 0.05$	7.6%	15.1%	23.4%	56.5%	20.1%	12.0%	3.2%

Table 1-3. The effect of λ_{addr} on the approximate percentage of clusters whose sizes are in certain ranges.

sizes are in certain ranges for the various values of λ_{addr}. This table further accentuates that the range of cluster sizes, as well as the density of cluster sizes at the extremes, increases as λ_{addr} decreases.

Table 1-4 shows a table comparing homogeneity when Linux was compiled with different compilers, compilation settings, and values of λ_{addr}. Each homogeneity value in this table is calculated as the median of 3 trials. The main insight to be gained from this table is that, compared to varying λ_{addr}, varying the compiler and optimization settings has a relatively small effect on the homogeneity of the resulting clustering.

Finally, we analyze the effect of λ_{addr} on the distribution of a directory's functions over clusters. If our technique worked perfectly, all functions in a directory would be placed in the same cluster. Unfortunately, it is most common for the functions in a directory to be placed in multiple clusters. Figure 1-7 shows the distribution of the number of clusters into which functions from a single directory are placed. The most noticeable feature of this figure is that directories are divided into fewer clusters as λ_{addr} increases. Figure 1-8 provides a dual perspective; this figure shows the

	gcc			clang		
λ_{addr}	-Os	-O2	-O3	-Os	-O2	-O3
60.0	0.819	0.822	0.824	0.820	0.815	0.819
1.0	0.627	0.644	0.605	0.637	0.613	0.613
0.05	0.426	0.410	0.397	0.414	0.399	0.393

Table 1-4. The effect of compiler, optimization flags, and λ_{addr} on the homogeneity of the clusters found by performing Louvain clustering on the combined graph constructed from the Linux kernel.

distribution of the number of directories represented by the functions in a cluster. Again, we see that as λ_{addr} increases, clusters tend to contain functions from fewer directories. Put together, these graphs suggest that as the layout of functions in address space becomes less important and the call graph becomes more important, it is more common to see clusters containing functions from many different directories. This observation is reflected in Figure 1-9 – this figure shows the distribution of address ranges in each cluster. Given that gcc and clang largely preserve function locality, clusters that contain functions from many different directories will contain functions from different parts of address space.

One might assume from the results of this section that address locality is much more important than the call graph. The following section shows why this is not always the case.

1.4.4. Qualitative validation – kernel function tracing

In this section, we provide a focused assessment of our technique that does not rely on the homogeneity score. In particular, we validate our cluster results by exploring how well our technique extracts and isolates the tracing system built into the Linux kernel. Tracing refers to the use of logging to record information about a program's flow of execution [23]. For example, one can use tracing to track system calls or kernel function calls in order to debug, profile, or otherwise assess the performance of the kernel (or user programs that make extensive use of kernel functionality).

Linux's implementation of tracing is rather intricate, utilizing a complex system of macros whose usage is not isolated to one directory. Furthermore, the directory that defines the plurality of tracing functionality, `include/trace/events`, defines functions that are scattered across 118 address ranges. As such, the tracing module is a prime example of a module that largely defies our assumption that the compiler preserves function locality. Because this module is defined across a large number of address ranges, it will not be captured well by our technique when we use larger values of λ_{addr} (i.e., values that typically provide higher homogeneity scores). Furthermore, because the tracing "module" is defined in multiple directories, we cannot even use the directory structure as ground truth.

We thus examine the tracing module by taking as ground truth 6 (partially overlapping) sets of tracing-oriented functions as defined in Table 1-5. This table also shows the extent to which these sets of functions do not adhere to our function locality assumption. Note in particular that sets 2

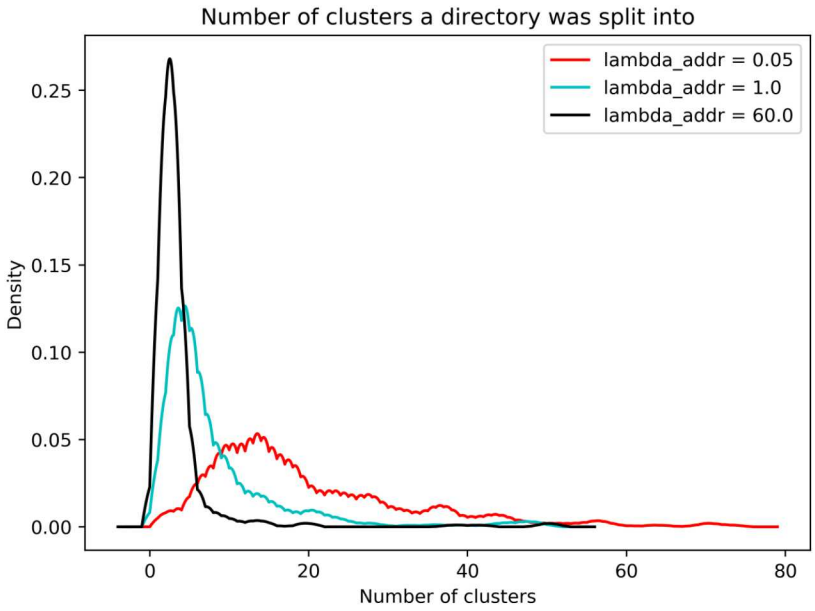

Figure 1-7. Kernel density estimates showing the distribution of the number of clusters a directory was divided into. For this figure, we consider only directories containing at least 20 functions.

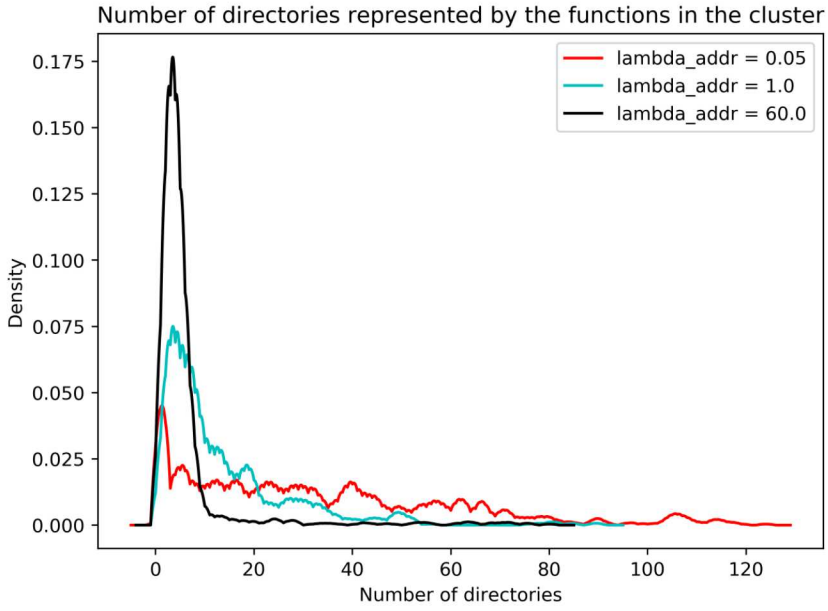

Figure 1-8. Kernel density estimates showing the distribution of the number of directories represented by the functions in a cluster. For this figure, we consider only directories containing at least 20 functions.

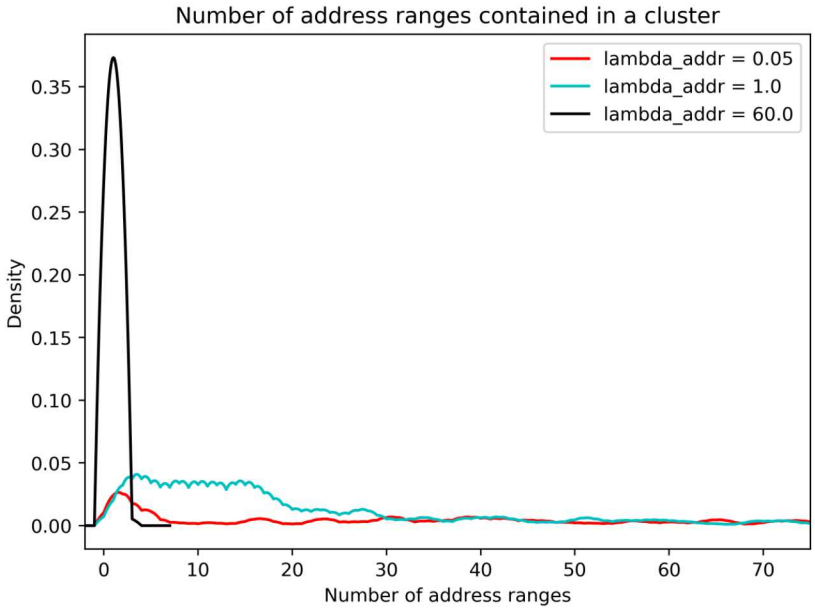

Figure 1-9. Kernel density estimates showing the distribution of the number of address ranges in each cluster. The x-axis of this figure was truncated at 75**.**

through 4 are each defined across 18 or 19 directories, and that all sets are segmented into over 80 address ranges. Table 1-6 shows how well our technique extracts each of the 6 defined sets of functions for various values of λ_{addr}. Note that all considered sets of tracing-oriented functions were clustered much more tightly for lower values of λ_{addr}. For example, when $\lambda_{\text{addr}} = 0.05$, over 98% of functions in sets 2 through 4 were found in one cluster, whereas when $\lambda_{\text{addr}} = 60.0$, each of these sets of functions were separated into at least 47 clusters, with no cluster having more than 18% of the functions in the set.

Ultimately, this analysis shows that the value of λ_{addr} must be chosen to suit the need or application at hand. Whereas high values of λ_{addr} are valued when the functions belonging to modules of interest are expected to be adjacent to one another in address space, lower values of λ_{addr} are recommended when the functions of interest are expected to be spread out in address space.

1.5. Discussion

In designing our experiments, we made two choices about how to evaluate our method. The first was that the directory structure of the original source code can be used as ground truth. The validity of this choice rests on the (not always valid) assumptions that the underlying source code was well organized in its directory structure, and that code used frequently in the same context is placed in the same directory. The second choice was to use homogeneity as an evaluation metric. Most metrics used to measure clustering take into account both the homogeneity (i.e., whether items in the same cluster belong together) and the completeness (i.e., whether all items that

Set #	Definition	# addr rngs	pct lrgst addr rng	# dirs	pct lrgst dir
1	Functions in `include/trace/events`	116	12.7%	1	100%
2	Functions beginning with `trace_raw_output`	87	17.2%	18	54.3%
3	Functions beginning with `trace_event_raw`	134	17.3%	19	54.1%
4	Functions beginning with `perf_trace`	122	10.2%	19	53.7%
5	The union of sets 2 through 4	134	17.4%	20	54.0%
6	The union of sets 1 through 4	141	17.3%	20	54.1%

Table 1-5. Six sets of functions of the Linux kernel that relate to tracing. For each set of functions, we provide a set number, the definition of the set of functions, the number of address ranges for the set, the percentage of functions in the set defined in the largest address range, the number of directories that overlap nontrivially with the set, and the percentage of functions in the set defined in the directory that contains the plurality of the set's functions.

	$\lambda_{\text{addr}} = 0.05$		$\lambda_{\text{addr}} = 1.0$		$\lambda_{\text{addr}} = 60.0$	
Set	# clstrs	pct lrgst	# clstrs	pct lrgst	# clstrs	pct lrgst
1	12	65.3%	40	55.3%	39	19.0%
2	3	99.2%	32	78.8%	47	17.4%
3	6	99.1%	29	91.7%	49	17.4%
4	8	98.1%	34	79.1%	50	17.3%
5	12	65.8%	43	59.1%	51	17.4%
6	17	65.6%	46	59.0%	52	17.3%

Table 1-6. Performance metrics for six sets of tracing-oriented functions. For each value of λ_{addr}, we report the number of clusters containing functions in the set and the percentage of functions in the set belonging to the clusters containing the plurality of the set's functions.

belong together are placed into the same cluster) of the clustering. We argue that, for our use case, completeness is much less important than homogeneity. Although it will take time to determine the degree to which a cluster is of interest to an analyst, regardless of the method used, it is perhaps not as big of an issue if, for example, twice as many such evaluations need to be performed than if the clusters contain unrelated functions. As future work, it would be beneficial to explore automated ways to extract or highlight clusters that would be of interest given a specific use case or analyst question.

Furthermore, it may be prudent to reexamine the mathematical interpretation of modularization used in this report (i.e., graph modularity) to see if our formalization maps well to real-world use cases. In particular, we rely heavily on the assumption that modules can be interpreted as groups of functions that interact with one another much more than with functions of other modules. This may not always be the case. For example, one might imagine a "math" library that contains implementations of a variety of unrelated mathematical functions. In this example, the module consists of a set of similar functions, but the functions of interest rarely call each other. Our approach was built under the hypothesis that this sort of module is less common than modules that are highly connected by their call graphs, and that such modules are not frequently the targets of interest of an analyst. We leave a characterization of different types of modules seen in binary code for future work.

As mentioned previously, we assume that our code was compiled without any control flow structures like virtual tables or indirect calls (i.e., we ignore virtual tables and indirect calls in our analyses). We expect that our general technique can be easily modified to incorporate these control flow structures. For example, it is almost guaranteed that two functions that appear in the same virtual table will belong to the same module, and indirect calls can be resolved to some extent using static analyses like static taint analysis, constant propagation, or symbolic execution. We may also be able to track object data when objects are passed between functions in order to resolve virtual table pointers.

In our experiments, we use Ghidra to extract functions. However, we do not have perfect knowledge of the performance of Ghidra when faced with non-debug and stripped binaries. Ghidra identifies fewer functions when faced with such binaries, but the severity of this problem is not well-understood. We also do not know how well Ghidra extracts global or read-only data accesses from non-debug and stripped binaries. Before our approach can be made fully operational, it would be prudent to assess the performance of Ghidra on non-debug and stripped binaries, and we may need to construct a more reliable method of detecting and extracting function information.

Furthermore, type information is lost when debug information is not available. Note that we did not use type information in this analysis, but it is possible that more complete knowledge of type information could be used to improve our techniques. We may be able to track allocation sites and identify which functions those allocations are passed to. Alternatively, patterns of data accesses that are offsets from pointers (structure members) might provide evidence that two functions manipulate the same type. For programs exhibiting a large degree of modularization, this could provide evidence that two functions belong to the same module.

Although we make ample use of control flow structures between functions (e.g., the call graph), we might be able to incorporate inter-function control flow information to identify functions belonging to the same module. Functions in the same module may have bits of code that have similar functionality. We may be able to group together functions whose control flow graphs share many common subgraphs.

It is also possible that we could augment our approach by incorporating data flow or patterns of data access into our approach. One might expect functions in the same module to reference the same flavor of data; this information could be incorporated into our existing approach as another indication that two functions belong to the same module. It should be noted that we experimented briefly with this idea. In particular, we tracked functions' use of global data and constructed a graph that connected two functions with an edge if they referenced the same global data address. In our experiments, however, incorporating global data references in this manner caused a decrease in homogeneity.

It should also be noted that we explored two other techniques for clustering functions. In the first, we performed a spectral clustering on the columns of the adjacency matrix of the combined call / address similarity graph. In the second, we constructed an unsupervised graph neural network. This network associated a vector to each node; our hypothesis was that the vectors associated to a node n's neighbors could be used to reconstruct the vector associated to n. Traditional clustering algorithms were then used to cluster the vectors. In our experiments, the spectral clustering and graph neural network approaches produced homogeneity scores that were approximately 35% and 60% of that of the modularity approach, respectively. Given these results, we did not work to refine these approaches after our initial experimentation; further experimentation may be warranted.

REFERENCES

[1] GHIDRA. https://ghidra-sre.org. Accessed: 2020-08-13.

[2] radare. https://rada.re/n/. Accessed: 2020-08-13.

[3] Rosetta's way back to the source. https://www.cs.vu.nl/~herbertb/projects/rosetts/. Accessed: 2020-08-13.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[5] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[7] Saed Alrabaee. *Efficient, scalable, and accurate program fingerprinting in binary code*. PhD thesis, Concordia University, 2018.

[8] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[9] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Connallen, and Kelli A. Houston. Object-oriented analysis and design with applications. *ACM SIGSOFT software engineering notes*, 33(5):29–29, 2008.

[10] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. Maximizing modularity is hard. *arXiv preprint physics/0608255*, 2006.

[11] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 129–143. Springer, 2006.

[12] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46. IEEE, 2005.

[13] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 673–674, 2006.

[14] Frank Eichinger, Klemens Böhm, and Matthias Huber. Mining edge-weighted call graphs to localise software bugs. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 333–348. Springer, 2008.

[15] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[16] Alexander Gounares, Christopher Warwick Fraser, and Steven Craig Venema. Methods and systems for binary scrambling, 2018.

[17] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. Code vectors: understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 163–174, 2018.

[18] Hex-Rays. IDA Pro. `https://www.hex-rays.com/products/ida/`. Accessed: 2020-08-13.

[19] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[20] Zhenhao Luo, Baosheng Wang, Yong Tang, and Wei Xie. Semantic-based representation binary clone detection for cross-architectures in the internet of things. *Applied Sciences*, 9(16):3283, 2019.

[21] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, pages 383–392, 2009.

[22] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

[23] Sergio Prado. Tracing the Linux kernel with ftrace. `https://embeddedbits.org/tracing-the-linux-kernel-with-ftrace/`. Accessed: 2020-08-27.

[24] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, pages 410–420, 2007.

[25] Vincent A Traag, Ludo Waltman, and Nees Jan van Eck. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific reports*, 9(1):1–12, 2019.

[26] Shuai Wang, Pei Wang, and Dinghao Wu. Semantics-aware machine learning for function recognition in binary code. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 388–398. IEEE, 2017.

[27] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 708–719, 2016.

[28] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[29] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1145–1152, 2020.

[30] Gang Zhao and Jeff Huang. DeepSim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 141–151, 2018.

DISTRIBUTION

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop
1	D. Chavez, LDRD Office	1911	0359

Email—Internal (encrypt for OUO)

Name	Org.	Sandia Email Address
Jacek Skryzalin	05646	jskryza@sandia.gov
Daniel Chivers	05638	dchiver@sandia.gov
Douglas P. Ghormley	05600	dpghorm@sandia.gov
Michelle A. Leger	05636	maleger@sandia.gov
Nasser J. Salim	05631	njsalim@sandia.gov
William A. Zortman	05646	wzortm@sandia.gov
Katherine M. Simonson	06000	kmsimon@sandia.gov
Technical Library	01177	libref@sandia.gov

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.