An Information Retrieval Process to Aid in the Analysis of Code Clones

Robert Tairas · Jeff Gray

Abstract

The advent of new static analysis tools has automated the searching for code clones, which are duplicated or similar code fragments in a program. However, clone detection tools can report many clones if the source code that is being searched is large. Programmers may have difficulty comprehending the extensive results from the detection tool, which may inhibit the ability to maintain the identified clones. Latent Semantic Indexing (LSI) is an information retrieval technique that attempts to find relationships in a corpus based on the analysis of the documents in the corpus and the terms in the documents. In this paper, LSI is used to cluster clone classes that have been identified initially by a clone detection tool. The goal of this paper is to detect trends and associations among the clustered clone classes and determine if they provide further comprehension to assist in the maintenance of clones. Experimental evaluation of the approach is reported from a sequence of tools that are chained together to perform an analysis of clones detected in the Microsoft Windows NT kernel source code.

Keywords: Information retrieval; Program comprehension; Latent semantic indexing; Code clone analysis

1. Introduction

Code clones are sections of source code that are duplicated in multiple locations in a program. Clones are generated mostly due to the copy-and-paste activity of programmers where one section of code is copied and pasted into another location, in some cases with changes and in other cases with no changes between the original and cloned code. The existence of code clones presents a challenge during software maintenance – for each fragment of code that is modified, the equivalent clones in other source parts of the program must often be updated accordingly. Providing a means to detect clones automatically allows the maintainer of the program to realize the existence of clones in the program. Furthermore, providing a process that offers additional information about the clones after their detection can be used in the understanding of the identified clones to better inform maintenance tasks.

Robert Tairas (✉) · Jeff Gray
Department of Computer and Information Sciences, University of Alabama at Birmingham,
1300 University Boulevard, Birmingham, AL 35294
e-mail: tairasr@cis.uab.edu

Jeff Gray
e-mail: gray@cis.uab.edu

Over the past decade, static analysis tools and techniques have been proposed to allow the automated detection of code clones in software, as seen in the Clone Detection Literature web site[1]. These tools have the ability to detect clones in large software systems, such as JDK 1.4.2, which has 2,418K lines of code (LoC) (Jiang et al. 2007) and the Linux kernel, which has 4,365K LoC (Li et al. 2004). A distributed version of a clone detection tool was able to detect clones using 80 computers in about 400 million lines of source code for a collection of software used for FreeBSD (Livieri et al. 2007a).

The ability to detect clones in large software systems in turn will yield detection results that are considerable in size. The number of clones that are detected will typically increase as the number of lines that are evaluated rise. The number of clones may become a hindering factor in the subsequent analysis of the clones. During analysis, each clone is analyzed to determine why it exists and what actions could be performed to reduce maintenance problems (i.e., what can be done to the clones to help improve the quality of the code and maintain the clones in the future). This can mean the knowledge of what certain clones do and where they are located, or the process of refactoring certain clones into abstractions where maintenance can be performed more easily. However, programmers may have difficulty to perform these tasks in terms of comprehending clones if the number of clones reported by a clone detection tool is large. From the examples stated previously, the number of cloned lines discovered by Jiang et al. in the JDK started from 204K LoC, which is 8% of the total lines of code. Li et al. found 122K LoC of what they called *copy-pasted segments*, which represents 15% of the total lines of the Linux kernel code. Both results illustrate the large amount of code identified as clones that can be returned by a clone detection tool that would consequently be part of the manual analysis process.

1.1 Existing Techniques for Comprehending Code Clones

Several techniques have been proposed to aid programmers in the comprehension of the large number of clones that may be detected in a program. These techniques can be divided into two groups: *classification* and *visualization*. Classification techniques partition clones into different categories based on properties of the clones. A set of clones can be classified based on where each clone is located with respect to the other clones in the hierarchy of files and directories for procedural languages (Kapser and Godfrey 2004) and in the type hierarchy for object-oriented languages (Koni-N'Sapu 2001). Clones can also be classified based on how different they are. A well-known classification of clones is based on three types: clones that exactly match each other, clones that differ only by the value of their identifier names (i.e., parameterized clones), and clones that are near-exact matches of each other (i.e., based on the addition and deletion of a few lines of code in the clones) (Bellon et al. 2007). Clones can also be classified based on whether the types used are different (Balazinska et al. 1999).

Various techniques have been used to visually represent clones. These include scatter plots in tools such as Duploc (Rieger and Ducasse 1998) and CCFinder (Kamiya et al. 2001), duplication web and system model views in (Rieger et al. 2004), clone system hierarchical graphs in (Jiang and Hassan 2007), and an aspect browser-like view in (Tairas et al. 2006).

The classification and visualization techniques just described can provide for a better understanding of the clones. Clones that are similar in syntax and structural properties are

[1] Clone Detection Literature web site, http://www.cis.uab.edu/tairasr/clone/literature

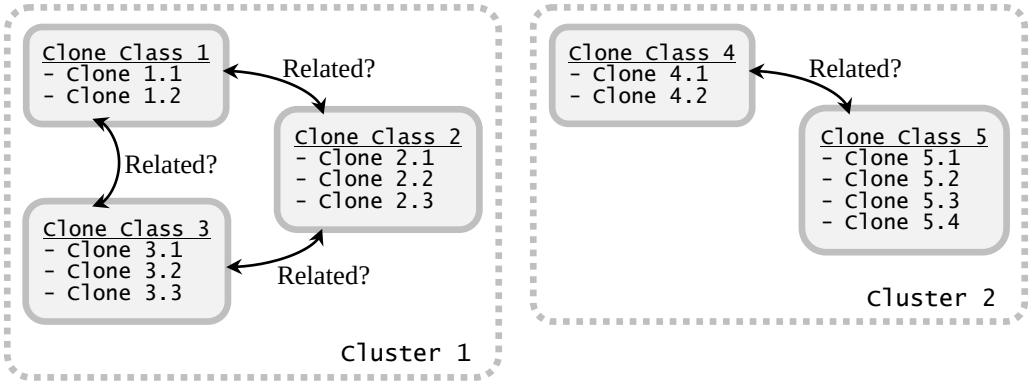
grouped together through these classification techniques. Subsequent understanding of the clones is based on the relationships among these classified clones and also the overall arrangement of the classification. However, these classification techniques look at just the syntactic and structural similarity properties of the clones. Other properties that are not necessarily syntax- or structure-based can potentially yield further relationships among the clones that can complement the classification techniques above. As for the visualizations, they tend to give more of an overall understanding of the clones with respect to the source code as a whole or a section of the source code and are less concerned with the specific content of the clones. An alternative method is needed to provide the grouping of clone classes to further aid in their comprehension.

1.2 An Information Retrieval Perspective on Code Clone Analysis

The field of information retrieval focuses on the extraction of useful information from a corpus of data. One specific technique in the area of information retrieval is Latent Semantic Indexing (LSI), which is concerned with obtaining relationships among the documents that divide a corpus and the terms in these documents. The relationships are discovered through the analysis of the terms and documents using matrix manipulation techniques (Deerwester et al. 1990). This technique has already been used in relation to clone detection (Marcus and Maletic 2001), where high-level concept clones were detected based on the identifiers and comments of the source code. However, this work does not incorporate syntax-based clone detection and thus the clones that are identified vary widely syntactically and are more conceptual in nature. The key questions addressed in this paper are: What if LSI is used on the clones that are initially detected by a syntax-based clone detection tool? Could LSI be used to group clone classes (detected by the clone detection tool) that it considers related to each other? Furthermore, could these groups aid in the comprehension of the clone classes, and in turn, the clones that are associated to them? The hypothesis here is that LSI can reveal relationships among clone classes that are not based on similarities in their syntax and these relationships can further assist in the understanding of the initial clones that were identified in a first-pass by a traditional clone detection tool.

This paper outlines an effort to determine the benefits of using LSI on the results of a clone detection tool to aid in the understanding and analysis of clones in the Microsoft Windows NT kernel source code. The definition of terms and documents in LSI depends on the domain that it is being used. In this case, the terms are the identifier names of the source code that is represented by the clones and the documents are the clone classes that contain the clones. LSI provides the ability to determine term-term, term-document, and document-document similarities. The objective in this paper is to determine the similarities between the documents (i.e., the clone classes). Clone classes that are similar are clustered together and these clusters are analyzed to determine whether they are useful towards the comprehension of the clones. Figure 1 illustrates the hierarchy of the clones, clone classes, and clusters. The clones in a clone class represent duplicated code that is considered similar to each other based on a specific similarity property. These clone classes are in turn clustered to determine relationships among the clone classes, where each clone class contains different groups of duplicated code (or clones). Several relevant tools will be chained together to produce the necessary output for the analysis of the clones.

Fig. 1 Hierarchy of clones, clone classes, and clusters



1.3 Corpus Description and Analysis Goals

The clones that comprise the corpus in this paper are obtained from the source code of the Windows Research Kernel 1.0, which is written primarily in the C language and has been made available by Microsoft to academia for teaching and research[2]. Implementations of basic operating system functions (e.g., process and thread support, I/O and memory management) for the Windows NT kernel are included in the distribution. Some parts that are not included in the research kernel are plug and play, power management, and a virtual DOS machine.

The goal of this paper is to determine the benefits of clustering clone classes based on the LSI information retrieval-based technique for comprehension purposes. To achieve this, we forego the clustering of all the clone classes at once, but instead divide the clustering and analysis of the clone classes into specific directories. In the NT kernel, the source files representing major OS functionalities are grouped into separate directories. We assert a hypothesis that more interesting clones will be found in specific directories that represent focused OS functionalities. The top-five directories in the NT kernel source containing the most clone classes were selected for this study. To further constrain the clones that are specific to one directory, the clone classes that were selected contain the property that all the clones in the class are located in the same directory. An additional goal is aimed at combining the clone classes from all five directories and clustering them to determine any relationships among the clones from these separate directories. To summarize, we want to see if the clustering of clone classes using LSI aids in the comprehension of the clones in specific directories separately and also in combination. From the above description, the scope of this work encompasses the sections of the Windows NT kernel code that have been determined to be clones. More specifically, the clones that exist in five directories in the NT kernel will be considered as input to the process that represents the contribution of this paper.

[2] Windows Research Kernel, http://www.microsoft.com/resources/sharedsource/Licensing/researchkernel.mspx

1.4 Outline and Structure of Paper

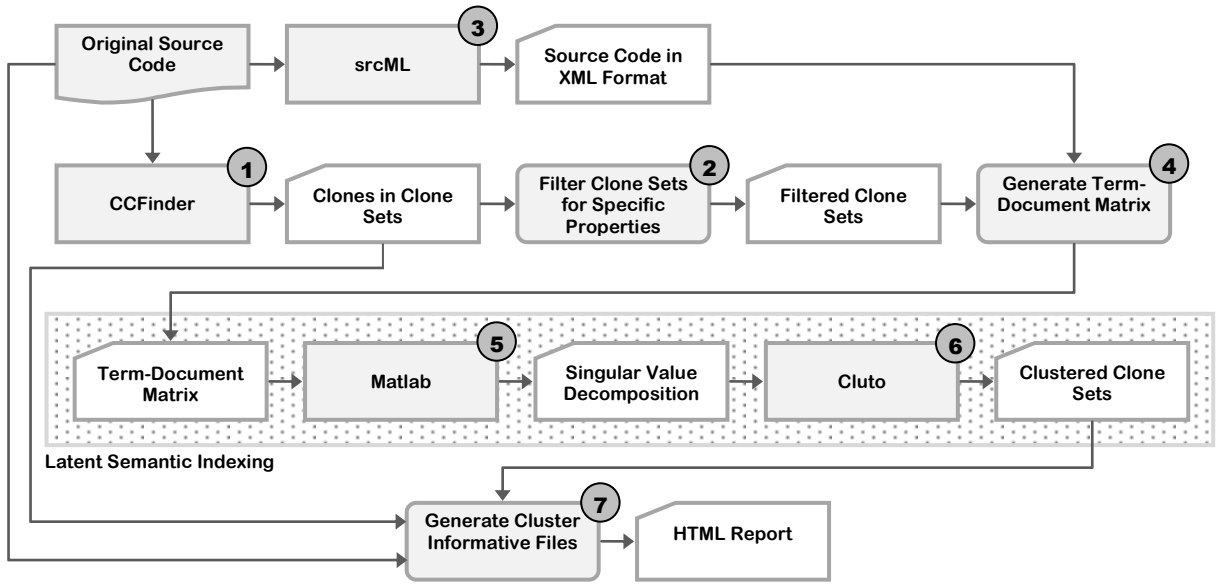
The remainder of this paper is structured as follows: the next section gives an overview of the process that is used to obtain and report the clusters of clone classes. Section 3 presents the results of the analysis of the clusters. Section 4 offers a discussion of the results from Section 3. This section also presents the benefits of using LSI on clone detection results in addition to discussions about the advantages and limitations of the process. Section 5 lists related work and Section 6 offers concluding remarks and points toward future work.

2. Clone Class Clustering Process

Figure 2 provides an overview of the process described in this paper. The source code is first searched for clones using a traditional clone detection tool. Then the identifier names contained in the source ranges of pre-selected clones are extracted and a term-document matrix is generated. The singular value decomposition (SVD) of this matrix is computed and the document-document similarity is used to cluster clone classes for further manual analysis. This process is described in detail in the following subsections.

Several third-party tools (represented by rectangles in Figure 2) were used to provide the necessary computations. In addition to these tools, some helper programs (represented by rounded rectangles in the figure) were written in Java to enable the integration of each tool. These helper programs are described throughout this section.

Fig. 2 Overview of the clone class clustering process



2.1 Clone Detection (Step 1)

A clone detection tool called CCFinder (Kamiya et al. 2002) was used to search for the initial clones in the NT kernel. CCFinder uses the token-based representation of the source code to find clones. It is freely available from the Software Engineering Laboratory at Osaka University[3]. Several different clone detection techniques and tools have been developed, but there has not been one clear choice that provides the *best* clone detection results. This is confirmed in the comparison of several clone detection techniques in (Bellon et al. 2007). CCFinder was selected more because of its availability (i.e., freely available) and ability (i.e., can detect clones for source code in the C language) and not for any specific output properties of the clone detection process. The technique introduced in this paper is open to results from other clone detection techniques. We have also performed clone detection using CloneDR (Baxter et al., 1998), but in this paper we limit the discussion to CCFinder as our single detection tool.

CCFinder provides a textual representation of its results where clone classes are associated with the token ranges of a clone pair. A clone pair consists of two sections of code that CCFinder has determined to be similar. A clone class is derived from the maximal set of connecting multiple clone pairs. Listing 1 is an example of the textual representation of clones given by CCFinder. The first line represents a clone class (ID: 262), which consists of a clone in one file (ID: 46) with token ranges 1806 to 1857 and a clone in another file (ID: 72) with token ranges 62-113. Similarly, the second line represents a clone class (ID: 1471), which consists of a clone in one file (ID: 47) with token ranges 1716-1768 and a clone in another file (ID: 13) with token ranges 9381 to 9433. The listing of this information is sorted by the file ID of the first clone in the clone pair (i.e., in Listing 1, a clone pair associated to file ID 46 is followed by a clone pair associated to file ID 47). These token ranges are converted into line numbers, which will be used to extract the identifier names associated with the source code inside the range of the clones. The line numbers are obtained from a file that CCFinder generates for each source file that has been searched for clones. This file contains one line per token with the line containing that token's information. The project web site supporting this paper [4] contains the CCFinder textual representation output file for the NT kernel source code (with file names removed).

Listing 1 Sample CCFinder textual clone representation

```
...
clone_pairs {
...
262    46.1806-1857 72.62-113
1471   47.1716-1768 13.9381-9433
...
}
```

In order to retrieve the clones for analysis in the multiple experiments described in this paper, the clone information is stored in a database. The database used is Apache's Derby[5], which is a Java-based open source relational database. The information for each individual clone is stored in the database with its association to a clone class.

[3] CCFinder, http://www.ccfinder.net
[4] Project web site, http://www.cis.uab.edu/tairasr/ir4pc
[5] Apache Derby, http://db.apache.org/derby

2.2 Clone Class Filtering (Step 2)

The analysis of the clones using LSI is focused on clones that are found in specific NT kernel directories. In particular, we look at the clones in five directories containing the most clone classes. LSI will be performed separately on the clone classes of each of these five directories. Table 1 displays the top-five directories with their clone class totals. The decision to focus separately on each of these five directories was made due to the fact that the clones that span multiple directories were observed to be few among the clone classes in these directories. That is, the majority of clone classes in the five directories contained clones that reside in the same directory. This can be seen in the difference of the total clone classes in the "All occurrences" column and the "Solely in directory" column in Table 1. A total of only 60 clone classes contained clones spanning multiple directories.

Table 1 Clone class totals of top-five directories in the NT kernel

Directory	All occurrences	Solely in directory	After subsets removed
Memory management	383	361	335
Registry configuration	213	209	196
Process/thread support	175	157	132
Security functions	155	148	121
I/O management	154	145	138

The totals under "All occurrences" represent the number of clone classes that contain at least one clone in the directory. Because the clustering process calls for clone classes with clones that are all contained in one directory, these clone classes are determined and the totals under "Solely in directory" represent these clone classes. Finally, it was observed that some clone classes were contained in other clone classes. That is, if two clone classes are defined as $CS_1 = \{c_1, c_3, c_5\}$ and $CS_2 = \{c_1, c_3\}$, then $CS_2 \subseteq CS_1$ and the clone class subset CS_2 is removed from further consideration. A clone class is also removed if the clones in that class are contained in the clones of another class. In this case, the clones are not exactly equal to each other, but the section of code representing the clones in the class that is removed is covered fully by larger clones in another class. Figure 3 provides examples of such eliminations. In Example 1, Clone Class 836 is removed, because the exact same clones are located in Clone Class 831. In Example 2, Clone Class 1734 is removed, because its clones are contained within the clones in Clone Class 1735. The totals under "After subsets removed" represent clone classes with clones that are all contained in one directory *and* are not subsets of any other clone class. Thus, the clone classes representing the totals from the last column are used in the clustering process.

Fig. 3 Examples of eliminated clone class subsets

Example 1

```
Clone Class 836
- File: psquery.c (lines 849 - 874)
- File: psquery.c (lines 971 - 998)
```

```
Clone Class 831:
- File: psquery.c (lines 849 - 874)
- File: psquery.c (lines 971 - 998)
- File: psquery.c (lines 1354 - 1376)
```

Example 2

```
Clone Class 1734:
- File: tokenset.c (lines 911 - 924)
- File: tokenset.c (lines 936 - 948)
```

```
Clone Class 1735:
- File: tokenset.c (lines 911 - 928)
- File: tokenset.c (lines 932 - 948)
```

2.3 Source Code XML Representation (Step 3)

In conjunction with the detection of clones, a tool called srcML[6], which is a freely available tool developed by the Software Development Laboratory at Kent State University, was used to convert the NT kernel source code into an XML representation containing selected abstract syntax tree (AST) tags. This representation is used to extract the identifier names that will be needed in order to generate the term-document matrix. The comments are not considered for the process in this paper because we would like to see how well the relationships of the clones are through the clustering without comment information.

Table 2 shows an example of the XML representation of the source code. The values of the elements of type *name* are of interest. An srcML XML representation of each source file in the directories that are analyzed is generated. Given the source code ranges (starting and ending line numbers) of the clones in the clone classes that were determined in Step 2, the identifier names that are contained in these ranges are extracted from the XML representations. This is possible because the original source code layout is preserved in the XML representation; as such, a line number in the original code corresponds to the XML representation for that same line in the XML file. No special filtering of names is performed. All values from elements of type *name* are included; hence, no stop words are removed. Furthermore, stemming of the identifier names is not performed.

Table 2 srcML representation of simple declaration

Original code	XML representation
`int x = 3;`	`<decl_stmt><decl><type><name>int</name></type> <name>x</name> =<init> <expr>3</expr></init></decl>;</decl_stmt>`

[6] srcML, http://www.sdml.info/projects/srcml

2.4 Generating the Term-Document Matrix (Step 4)

The definition of *term* in this paper is the name of an identifier. The definition of *document* is a clone class. A term-document matrix is generated for each of the five directories. In addition, a final matrix is generated containing the clone classes and terms from all five directories. Table 3 provides totals of terms and documents for each matrix yielding the dimensions of the matrices. Please note that the number of terms in the last row is not the same as the sum of the previous five rows because an identifier name can exist and be counted in each directory, but when all directories are considered it is counted as a term only once. The number of documents in the last row equals the sum of the previous five rows, because the clone classes in each directory are not duplicated in other directories. Moreover, the document totals are identical to the clone class totals (last column) of Table 1.

Table 3 Term and document totals

Directory	# of terms	# of documents
Memory management	1506	335
Registry configuration	1035	196
Process/thread support	597	132
Security functions	670	121
I/O management	908	138
All five directories	4153	922

2.5 Computing the SVD of the Matrices (Step 5)

An important part of the LSI process is obtaining the SVD of the term-document matrix. An SVD of a matrix, say $X_{m\times n}$, produces three matrices; i.e., $X_{m\times n} = U_{m\times n} S_{n\times n} V_{n\times n}^T$, where S is a diagonal matrix containing singular values, which are ordered from largest to smallest along the diagonal. By selecting the first largest singular values, say k, a new matrix $\hat{X}_{m\times n}$ can be obtained; i.e., $\hat{X}_{m\times n} = U_{m\times k} S_{k\times k} V_{n\times k}^T$. This matrix is an approximate (but still valuable) representation of the original matrix X with the added property that the lower dimensionality removes some of the smaller components in the relationship (i.e., the noise). Given $\hat{X}$, cosine similarity calculations can be performed between the documents (in this case, the clone classes) that are represented by column vectors of $\hat{X}$ (Deerwester et al. 1990).

The commercial tool Matlab[7] was used to obtain the SVD of the term-document matrix and from this the generation of the lower dimensionality matrix that will be used to perform document-document similarity calculations, which is done in Step 6. The term-document matrix from Step 4 can be imported into Matlab and the SVD is generated using Matlab's standard `svd` function. Before this is done, the values in the matrix are normalized using the `zscore` function. Regarding the determination of the value for k or how many of the first largest singular values to include, we use the formula in (Kuhn et al. 2007), where $k = \lfloor m \times n^{0.2} \rfloor$.

[7] Matlab, http://www.mathworks.com/products/matlab

2.6 Clustering the Clone Classes (Step 6)

The clustering of the clone classes based on their similarities represented by the column vectors from the matrix $\hat{X}$ is obtained using Cluto[8], which is a clustering tool available from the Karypis Laboratory at the University of Minnesota. Cluto provides several clustering techniques that can be divided into *partitional* and *agglomerative*. Partitional clustering starts with one cluster containing all elements and iteratively partitions or divides the original cluster into smaller clusters. Agglomerative clustering is the opposite of partitional (Han and Kamber 2006). Cluto also provides several techniques for calculating vector similarities including calculating the cosine of two vectors, which is the similarity calculation that is used to generate the clusters in this paper.

The default clustering approach in Cluto is called *repeated bisections clustering*, which is a partitional clustering technique. In addition to the input matrix containing the vectors representing the clone classes, the only other argument requested by Cluto is the final number of clusters in which the clone classes will be grouped. Several values were tested for each of the directories. Two properties of the clusters were observed: the number of members in each of the clusters and the average similarity of the members of the clusters. The average similarity here is the average of all the similarity values that are calculated between the objects of a cluster. We look for a fairly uniform distribution of clone classes in each cluster, thus trying to avoid clusters that dominate other clusters or clusters that contain only one clone class. We observe the collection of the average similarity values from each clustering size that is generated and look for a considerable number of high average similarity values. This resulted in a cluster size of 30 for the first two directories (memory management and registry configuration) and a size of 20 for the last three directories (process/thread support, security function, and I/O management).

2.7 Generating Cluster Reports for Analysis (Step 7)

The result from Step 6 is a mapping of each clone class as a member of one of the generated clusters. At this point, the clusters need to be analyzed manually for trends and associations among the members of each cluster and in turn the clones of each clone class in the clusters. Informative HTML files containing all the source code in the clone classes that are contained in a cluster are generated from the results of Cluto, the results of CCFinder, and the original source code. The results of the analysis of the clusters are described in Section 3.

2.8 Scalability

With the exception of Matlab, the third-party tools used in the process outlined in this section are freely available. All of these non-commercial tools are scalable and can be used on the source code of large-scale software systems. As can be seen in (Li et al. 2004), CCFinder has been used to detect clones in the source code of the Linux kernel. The srcML translator can translate code into XML at a rate of 7500 LoC per second (Collard and Maletic 2004). Cluto has been used to

[8] Cluto, http://glaros.dtc.umn.edu/gkhome/cluto/cluto/overview

cluster large object sets including topic clustering of thousands of document datasets (Zhao and Karypis 2005). The main bottleneck is the final step of the process related to the manual evaluation of the generated clusters. This is a limitation in terms of scalability, which is described further in Section 4.4.

3. Analysis Results of the Clustering of Clone Classes within the Windows NT Kernel

Table 4 provides a statistical overview of the clusters that were separately generated from the clone classes of the top-five directories in the Windows NT kernel using LSI and a combination of all clone classes from the directories. The first column contains the number of clusters that were generated. The second column represents the number of clone classes for each clustering process of each directory after filtering is done to remove some clone classes as described in Section 2.2. The third column indicates the average number of clone classes in each generated cluster. The rest of the columns contain information about the average similarities (grouped by percentile ranges) among the members (i.e., clone classes) of the clusters that were generated. These values were reported by Cluto after the clustering was performed. For example, in the memory management directory, six clusters contained clone classes with similarity 90% and above, 13 clusters with similarity between 80% and 89%, and 11 clusters with similarity between 70% and 79%.

Table 4 Statistical information on the generated clusters

				Percentile range							
				90		80		70		60	
Directory	#oC	#oCC	AvgCC	#oC	%oC	#oC	%oC	#oC	%oC	#oC	%oC
Memory management	30	335	11.1	6	20%	13	43%	11	37%	0	0%
Registry configuration	30	196	6.5	11	37%	9	30%	9	30%	1	3%
Process/thread support	20	132	6.6	16	80%	4	20%	0	0%	0	0%
Security functions	20	121	6.0	10	50%	9	45%	1	5%	0	0%
I/O management	20	138	6.9	9	45%	10	50%	1	5%	0	0%
All five directories	50	922	18.4	10	20%	25	50%	13	26%	2	4%

#oC = Number of clusters
#oCC = Number of clone classes in directory
AvgCC = Average number of clone classes in each cluster
%oC = Percentage of clusters in each percentile range

A large number of the similarity values in Table 4 were 70% and above. The possible explanation for this is the nature of the clone classes reported by CCFinder. Two types of clone class combinations may have contributed to this observation:

- *Linked clone classes*: Two clone classes may be linked to each other by one or more clones that are present in both classes. For example, if two clone classes are defined as $CS_1 = \{c_1, c_2, c_3\}$ and $CS_2 = \{c_3, c_4, c_5\}$, then CS_1 and CS_2 will produce a very high similarity value.

- *Sliding clones*: The only difference between the clones in two clone classes is the slightly different starting and/or ending lines of the clones, which may amount to just a few lines.

The removal of clone class subsets described in Section 2.2 does not capture the two cases above. However, for both cases it was decided not to try to combine these clone classes together before generating the term-document matrix, because keeping these clone classes separated may yield interesting observations during analysis. Nevertheless, these two types of clone classes were still detected and reported in the cluster reports, so during the analysis the existence of such clone classes in the clusters can be noted. For sliding clones, clone classes with clones that differ with a maximum of five lines between the values of their starting and/or ending lines are associated with each other and noted in the report. This will allow the clones in the clone classes to differ by a few statements, but still be considered as representatives of the same clones.

The results of the analysis of the clustering of NT kernel clone classes are given in the following subsections, where the individual results from selected clusters are discussed. Each set of results is preceded with some statistical information about the cluster, such as the number of unique clone classes and the average similarity values among members of the clusters. Unique clone classes are those individual clone classes that cannot be grouped based on the two types that were previously described (i.e., linked clone classes and sliding clones). Clone classes that can be grouped based on these two types are counted as one unique clone class. The *Microsoft Windows Internals* book (Russinovich and Solomon 2005) provided a high-level understanding on how the code fits within the kernel operations. The results from each cluster data set are discussed separately. An overview and discussion of the entire analysis is given in Section 4.1. It is worth noting that the observations given in these subsequent subsections and the overview reported in Section 4.1 can be considered subjective based on our observations because the final step of the process requires a manual evaluation of the clusters.

3.1 Memory Management Directory

This sub-section presents four of the clusters observed in the memory management directory. The first cluster contains clone classes where the clones are logically opposite of each other. The clones in the second cluster utilize a similar structure of conditional statements. The third cluster reveals relationships based on the assignment of two variables, and the fourth cluster contains clones that revolve around a sequence of statements for initialization purposes.

Cluster MM-1 (Unique clone classes: 3 (out of 9 clone classes); Average similarity: 0.997)
All clone classes in this cluster reside in one file containing the implementation of memory page allocation. The clones in these clone classes account for ~230 cloned lines of code (CLoC) and represent a task that checks the availability of paged and non-paged memory pools and whether a low or high threshold has been reached. The signal state of a paged event is set when values surpass a threshold. Listing 2 provides pseudo-code of the two types of clones that are clustered together with differences indicated in bold. The operator in the conditional statement could be either a '<' or '>.' Based on this operator, the "High" or "Low" thresholds are checked. This sequence of statements is used in four different functions ranging from one to four occurrences in each function.

Listing 2 Pseudo-code of setting paged events

```
Free pools = Maximum (non) paged pools – Allocated (non) paged pools

if (Free pools (<|>) (High|Low) (non) paged pool threshold)
  if ((High|Low) (non) paged pool event is in signaled state)
    Set (High|Low) (non) paged pool event to nonsignaled state
  if (Free pools (<|>)= (Low|High) (non) paged pool threshold
    if ((Low|High) (non) paged pool event is in nonsignaled state)
      Set (Low|High) (non) paged pool event to signaled state
```

Cluster MM-11 (Unique clone classes: 8 (out of 12 clone classes); Average similarity: 0.854)
This cluster contains clone classes that in turn contain clones (~570 CLoC) with conditional statements (i.e., `if` and `switch` statements). These statements are primarily responsible for testing the property of a page mapping (e.g., cached, not cached, or write-combined) before executing. The clones are scattered among seven files in the directory. It is worth noting that two clone classes that should be included in this cluster are located in another cluster.

Cluster MM-13 (Unique clone classes: 9 (out of 14 clone classes); Average similarity: 0.758)
Except for five clone classes, this cluster includes clones (~910 CLoC) containing sequences of statements that obtain the base address of the allocated region of pages and size of the page region inside an exception handler. The two variable assignment statements inside an exception handler represent the duplication among the clones. Three different methods are used to obtain the two values as displayed in Listing 3. The clones are located in files related to the management of virtual memory (e.g., allocation, flushing, freeing, locking, and protection). The first method is used in six functions in five different files. The second method is used in two functions in one file. The third method is used in two functions in two files.

The first method is used at the beginning of a function to determine write access on the address that is passed. The last two methods are located at the end of a function and are used to update the value of the pointers. These clones were separated into the three clone classes, because of the difference in the structure of the assignments.

Listing 3 Three different methods of obtaining base address and region size

Method 1:

```
try {
  if (PreviousMode != KernelMode) {
    ProbeForWritePointer (BaseAddress);
    ProbeForWriteUlong_ptr (RegionSize);
  }

  CapturedBase = *BaseAddress;
  CapturedRegionSize = *RegionSize;
} except (ExSystemExceptionFilter ()) {
  return GetExceptionCode ();
}
```

Method 2:

```
try {
  *RegionSize = ((PCHAR)EndingAddress - (PCHAR)PAGE_ALIGN(CapturedBase)) + PAGE_SIZE;
  *BaseAddress = PAGE_ALIGN(CapturedBase);
} except (EXCEPTION_EXECUTE_HANDLER) {
  return GetExceptionCode ();
}
```

Method 3:

```
try {
  *RegionSize = CapturedRegionSize;
  *BaseAddress = StartingAddress;
} except (EXCEPTION_EXECUTE_HANDLER) {
  NOTHING;
}
```

Cluster MM-22 (Unique clone classes: 8 (out of 9 clone classes); Average similarity: 0.798)

Six out of the eight unique clone classes contain a sequence of statements that initializes a page and maps it with a page table entry. Most of the clones (~470 CLoC) in the cluster are scattered in four functions in three files and are called during system initialization. Different clone classes were detected possibly because of several reasons. The statements are used both in a one-time execution and in an iterative process inside a loop. Also, the code preceding this operation is not similar among the different clone classes. Finally, some clones utilize a function that performs both the initialization and setting the page table entry to a valid state. Other clones utilize a different function for initialization, while the page table entry is set to the valid state after the call to this function. Listing 4 displays three statements that are present in most of the clones, where the index of a page that contains all zeroes is retrieved based on a page coloring attribute (which is used to optimize cache behavior). The index is then assigned and initialized.

Listing 4 Example code for page initialization

```
DirectoryFrameIndex = MiRemoveZeroPage (MI_GET_PAGE_COLOR_FROM_PTE (StartPpe));
TempPte.u.Hard.PageFrameNumber = DirectoryFrameIndex;
MiInitializePfnAndMakePteValid (DirectoryFrameIndex, StartPpe, TempPte);
```

3.2 Registry Configuration Directory

The clusters generated from this directory are dominated by clone classes containing short sequences of code that reside mostly in one file. This is unlike the clusters observed in the other directories, where the clones in a clone class can be scattered around several files and contain slight variations. Because this directory is related to the Windows registry configuration, short sequences of duplicated code can be observed that are used to perform key look-up, setting of key values, and retrieving key values. Table 5 lists some examples of the clusters in this directory. A description of the code represented by the clones is given with the number of unique clone classes containing the clones. In addition, the number of files where these clone classes reside are included. It can be seen that the clones are uniquely contained in just one or two files.

Table 5 Clusters of registry functions and file totals

Code description of cluster	#oCC	#oFi
Comparison of values in a binary search	4(4)	1
Checking the consistency of the values in the registry list	3(4)	1
Setting value of a key	2(3)	2
Getting value of a node	2(8)	1
Getting value of a node	9(21)	1
Getting value of a node	11(14)	1

#oCC = Number of unique clone classes (total number of clone classes)
#oFi = Number of files containing clones

3.3 Process/Thread Support Directory

Two of the clusters observed in the process/thread support directory are introduced in this sub-section. Both clusters are associated by similar sequences of statements that are needed as part of a large sequence to perform different tasks.

Cluster PS-0 (Unique clone classes: 4 (out of 8 clone classes); Average similarity: 0.985)
Several tasks are performed that are based around a loop and initial statements inside the loop that iterate through an array of pointers to callout routines. The tasks include notifying the registered callout routines when an image, process, or thread is created and when a process or thread is deleted. Another task utilizes the same loop structure and statements to remove a single callout routine from the list.

Cluster PS-9 (Unique clone classes: 7 (out of 7 clone classes); Average similarity: 0.827)
Two of the clone classes in this cluster represent two pairs of function clones. The first pair consists of accessor functions to get and set the context of a thread. The second pair consists of functions to resume and suspend threads in a process. In both cases a similar sequence of statements is used to reference the thread or process object, perform an operation on the objects (e.g., get/set context, resume/suspend process), and generate a status value of the execution that is the return value of the function.

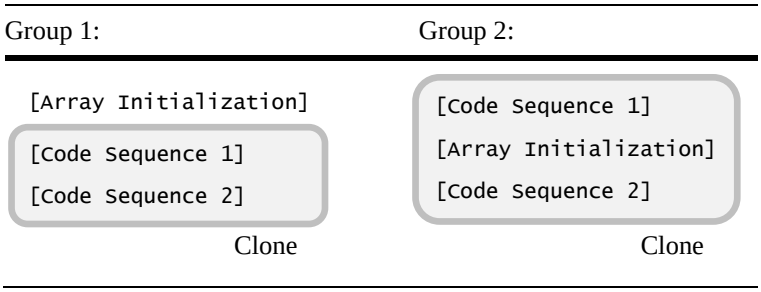
3.4 Security Functions Directory

The clusters related to the security functions directory contain similar properties to the clusters in the memory management directory. This can be observed from the first cluster being associated with a sequence of conditional statements where the clones differ in what is done inside the statements. Also, the clones in the third cluster are related by a variable assignment. The second cluster is based on clones where the statements are in different orderings.

Cluster SE-0 (Unique clone classes: 2 (out of 2 clone classes); Average similarity: 0.985)
This cluster represents two clone classes in two files that contain a sequence of conditional statements in which one sequence acts as a counter and the other copies information. Both files are related to the audit policy elements for security auditing purposes. The audit policy determines which properties should be recorded for future auditing. In the clone classes, each property is represented by a conditional statement. In the first clone class, the conditional statements are used to update a global counter of all tokens that have certain properties enabled. The second clone class uses the conditional statements to copy properties of a specific token to a token information variable. Further investigation revealed an additional sequence of similar conditional statements outside, but near, the clone ranges of one of the clone classes.

Cluster SE-7 (Unique clone classes: 2 (out of 4 clone classes); Average similarity: 0.992)
This cluster contains six unique clones located in six different functions in a file implementing security audit and alarm procedures. The four clone classes that comprise these clones can be divided into two unique groups, because of a slight difference in the syntax of the clones. All clones perform an initialization of the array that will contain audit parameters, but the location of this initialization is different for the two groups. Moreover, one group does not include the initialization code as part of the clones. Figure 4 illustrates the structure of the two groups.

Fig. 4 Differing location of array initialization



Cluster SE-11 (Unique clone classes: 11 (out of 13 clone classes); Average similarity: 0.917)
In this cluster, we see a pattern of code similar to cluster MM-13 where assignments are enclosed inside a `try` block for exception handling. However, in this case the code inside the `try` block is identical. The difference is the statement immediately preceding the block. Identical `try` blocks are separated into different clone classes based on the assignment statement preceding the `try` block. The clone classes all reside in the same file that returns information about a security token that is queried. The value that is assigned in the `try` block is the length of the token information that will be returned. The length is determined in three different ways in statements preceding the block, as seen in Listing 5. Not only are simple assignment statements used, but also one assignment statement is contained in a loop.

Listing 5 Three examples of obtaining token information length

```
Example 1:
RequiredLength = (ULONG)sizeof( TOKEN_STATISTICS );

Example 2:
RequiredLength = (ULONG)sizeof(TOKEN_GROUPS_AND_PRIVILEGES) +
                   PrivilegesLength + RestrictedSidsLength + GroupsLength;

Example 3:
while (Index < Token->RestrictedSidCount) {
  RequiredLength += SeLengthSid( Token->RestrictedSids[Index].Sid );
  Index += 1;
}
```

3.5 I/O Management Directory

From the observed clusters in I/O management directory, one cluster shows a similar finding to the related work of (Marcus and Maletic 2001). Four other clusters that are described in this sub-section contain clone associations based on specific variations in the code represented by the clones.

Cluster IO-0 (Unique clone classes: 5 (out of 5 clone classes); Average similarity: 0.961)
Similar to the results found in (Marcus and Maletic 2001), which detected implementations of linked lists, this cluster contains clone classes (except for two classes) related to code that implements and utilizes linked lists. One clone class represents function clones to add items to the head and tail of a list. Another clone class represents function clones that call list insertion functions. The final clone class contains a loop that calls the removal functions.

Listing 6 IRP allocation and initialization general code

```
// Allocate and initialize the I/O Request Packet (IRP) for this operation.
// The allocation is performed with an exception handler in case the
// caller does not have enough quota to allocate the packet.
irp = IoAllocateIrp( deviceObject->StackSize, (...) );
if (!irp) {
  //
  // An IRP could not be allocated.  Cleanup and return an appropriate
  // error status code.
  //
  if (...) {
    ExFreePool( event );
  }
  IopAllocateIrpCleanup( fileObject, (...) );
  return STATUS_INSUFFICIENT_RESOURCES;
}
irp->Tail.Overlay.OriginalFileObject = fileObject;
irp->Tail.Overlay.Thread = CurrentThread;
```

Cluster IO-5 (Unique clone classes: 16 (out of 24 clone classes); Average similarity: 0. 959)
This cluster contains 20 clones (~990 CLoC) in 20 functions in 11 files. The clones represent a sequence of code related to allocating and initializing an I/O Request Packet (IRP) as seen in

Listing 6. The highlighted parts of this code represent slight variations even in the comments area. Variations also occur in the parameters of two functions and the Boolean expression of a conditional statement, which are represented by "(…)."

Three other clusters displayed similar content where the clone classes represented code that contained slight variations. These clusters are listed in Table 6. Similar to the clones in the cluster of Listing 6, the clones in these clusters also reside in separate functions and in multiple files.

Table 6 Additional clusters containing clones with slight variations

Code description of cluster	#oCC	#oClo	#oFu	#oFi
Code to determine if an I/O operation on a file is synchronous and to wait until current thread owns the file	8(11)	16	16	10
Allocation of a memory description list (mdl), which is a specification of the physical memory occupied by a user's buffer when an IRP is created	4(4)	7	7	4
Code related to waiting for an I/O operation to complete	5(6)	8	8	6

#oCC = Number of unique clone classes (total number of clone classes)
#oClo = Number of unique clones
#oFu = Number of functions containing clones
#oFi = Number of files containing clones

3.6 All Five Directories

The clone classes of all five directories were clustered to determine any relationships among these clone classes as a whole. Almost half of the clusters that were generated contained clone classes from the same directories. Table 7 provides the number of clusters containing clone classes from exclusively one directory. This is not an unreasonable result as the clone classes that were clustered were chosen to be highly unique to the specific directories.

Table 7 Clusters where all clone classes are from one directory

Directory	30 clusters	50 clusters
Memory management	6	13
Registry configuration	2	4
Process/thread support	1	2
Security functions	2	3
I/O management	3	5
Total	14	27

Considering these clusters that contained clone classes from multiple directories, similarities of the code were mainly based on generic functions that appear across the directories. One cluster contained exception handling code covering a single variable assignment. Another cluster contained the frequently used `Status` variable that is the return value of many functions and used to determine the successful completion of a function execution. It was also observed that clusters were generated for code containing a `Process` variable that performs tasks on a process.

The observations from the clusters generated for each specific directory produced more interesting results in terms of clone location and functionality compared to the clusters generated from all five directories together.

4. Discussion of Analysis Results and Clustering Process

This section provides several discussion points. Section 4.1 evaluates the analysis outlined in the previous section and discusses the benefits of this information toward the comprehension of the clones. Section 4.2 describes the influence of utilizing the results from a clone detection tool, and Section 4.3 outlines the influence of using a specific clone detection tool. Observed limitations of the process described in this paper are given in Section 4.4.

4.1 Evaluation of the Cluster Analysis

A recurring theme from the analysis results is the discovery of sections of code that could be considered clones of each other, but were not determined by the clone detection tool to be so because of some statement variations. In cluster MM-1, slight changes were observed in the conditional operators that produced multiple classes of clones. In cluster MM-11, duplicate code was detected inside different conditional structures. In clusters MM-13 and SE-11, depending on the method of variable assignment a clone can be grouped with other clones. The clones in cluster MM-22 contain three statements that are present in most of the clones. The code surrounding these three statements differ, so in this case the clone classes in the cluster represent clones that are grouped based on the similar code that surrounds these three statements.

Variations are also discovered inside sequences of conditionals, loops, and functions in clusters SE-0, PS-0, and PS-9, respectively. In each case the statements inside the blocks are varied to perform specific tasks, but the target elements in which the tasks are performed are the same. In cluster IO-5, the grouping of clones is based on whether a conditional statement is present. Moreover, clones that are semantically the same can also be clustered as evident in cluster SE-7, where the only difference is the statements not being in the same order.

The observations of these noted clusters show associations among clones in the different clone classes. These associations are mainly based on similar code or functionality that includes some variations, which resulted in the clones being assigned different clone classes by the initial clone detection tool. With the analysis obtained from these clusters, the goal is to determine whether the results of this analysis provide a better understanding of the clones. The knowledge from the clustering observations can give the programmer an improved understanding of the clones, because he/she is exposed to the relationships of clones from multiple clone classes where the clone classes were originally separated by the clone detection tool. Comprehension in this case is related to the additional knowledge of how the code in clustered clone classes that differ by only slight changes in structure are used, such as varying assignment statements in clusters MM-13 and SE-11, different orderings of statements in cluster SE-7, and the existence of statements and differing function parameters in cluster IO-5. In addition, comprehension can be related to the knowledge of how sections of code that perform different tasks on an element or object in the program are associated to each other through the clusters, such as the various uses of code for page initialization in cluster MM-22, different tasks performed on an array of

pointers in cluster PS-0, and two uses of a sequence of conditional statements in cluster SE-0. Furthermore, knowing the location of related clone classes based on the clusters identified in Section 3 can help in the maintenance process, because clones that are related but are not exactly identical may still need to be updated at the same time or also possibly be refactored. For example, when the addition of a conditional statement is needed in one of the clone classes in either cluster MM-11 or SE-0, the knowledge of associated clone classes containing similar conditional statements becomes useful to determine whether or not the clones in these other clone classes should be updated as well.

Clones that dominate sections of code can also be identified, as seen in some of the clusters of generic functions in the clustering of all five directories and in some of the clusters in the registry configuration directory where the clones are uniquely located in one or two files. It may be argued that this information could be found in an IDE by searching for references to a function that surrounds a sequence of statements, but clone detection is needed to first determine which collection of statements are duplicated. Clustering then provides a method to group clones further to discover the functions that are used the most.

The utilization of the process described in this paper by a programmer in an effort to understand the clones of a program as compared to having only the results of the clone detection tool can been seen in this section. Without the additional associations provided through the clustering of the clone classes, those classes that are related but contain some variations thus being separated by the clone detection tool would not be known. This is in addition to the possibility of clone classes containing clones that are used in different contexts based on the code surrounding the clones. Without the clustering, these relationships would have to be determined manually by evaluating each of the clone classes. The process described in this paper also requires manual evaluation at the end, but the clustering of the classes provides some assistance during this evaluation by associating clone classes based on the information retrieval technique of LSI.

4.2 Influence of Clone Detection Results

The novelty of the process outlined in this paper compared to existing works that will described in Section 5 (i.e., (Marcus and Maletic 2001) and (Kuhn et al. 2007)) is the narrowing of the corpus from the entire source code to only sections of source code that have been identified by a clone detection tool. As stated throughout this paper, the purpose of narrowing the corpus is to determine associations among clone classes from the clone detection results. If the information from the clone detection results is removed and LSI is performed solely on the entire source code as the corpus, the determination of the definition of *documents* for the LSI process must be specified differently. The definition of *terms* for the LSI process would remain the same, namely the identifier names found in the code. Documents, however, would need to be specified as being an entire source file, a function block, or a block of statements. Based on the definition of documents, the clustering of these documents can reveal similar results as the clusters in this paper. For example, if the documents were set as function blocks, a cluster such as cluster IO-0 that represents function-level clones could be generated. However, in this case the clusters would only consist of function blocks. Statement-level clusters could only be generated if the documents were defined as statement blocks.

Focusing on sections of code that represent clones identified by a clone detection tool and specifying documents to be the clone classes containing these clones provides a clustering process where the clusters can ultimately consist of code of different levels. The clusters will consist of clone classes, which contain clones that can be statement-level clones, function-level clones or even file-level clones. The limitation of having to determine various definitions of documents in the LSI process is eliminated as the structure of clone classes is based on the syntactic similarity of different levels of code and in turn can consist of these different levels.

4.3 Influence of Chosen Clone Detection Tool

This paper utilizes CCFinder as the source for information of clones in the NT kernel source code. As seen in Section 2.1, CCFinder was chosen because of its availability and ability to detect clones in the NT kernel source code. The influence of this clone detection tool in the process outlined in this paper is considerable, because the terms and documents used in the LSI process is dependent upon the sections of code that have been identified as the initial clones by the clone detection tool. However, the sections of code representing the clones are exactly the part of the source code that is of interest in this paper rather than the entire source code altogether. As the clustering is dependent on the results of the clone detection tool that is used, the clustering will be different if another clone detection tool is used, because the results from various clone detection generally differ, as evident in (Bellon et al. 2007). Performing the process outlined in this paper with another clone detection tool is considered future work as described in Section 6.

Regarding the characteristics of a clone detection tool, Section 4.1 has given an evaluation of cluster analysis. The recurring theme that we observed was that the clones in the clone classes inside a cluster could have been lumped into one clone class, but because of some statement variations, the clone detection tool separated the clones into several clone classes. Clone detection tools try to return the maximal size of clones. For example, in clusters MM-13 and SE-11, because the different assignment methods are duplicated in several locations, the clones are grouped into classes based on the assignment method to produce the maximal size clones.

Another characteristic of a clone detection tool is the configuration setting that sets the minimum allowable clone size. The three statements evident in the clones in cluster MM-22 are considered lower than the minimum clone size and as several duplicate patterns exist around them, multiple classes of clones are generated. Determining the minimal size of a clone that can still be of interest is not an easy decision. Keeping the minimum size of clones to at least five lines will reduce the number of generally uninteresting clones (i.e., one line accessor methods). However, this will also separate the small clones with duplicate functionality such as those in cluster MM-22.

At the beginning of Section 3, two types of properties of the clusters (i.e., linked clone classes and sliding clones) were acknowledged. These properties are related to the resulting clone classes that were produced by CCFinder. It was decided that the classes containing these properties would be kept and not removed but noted during the evaluation of the clusters. These two properties did not strongly influence the observations of the clusters. However, the knowledge of the linked clone classes and sliding clones provided information about the uniqueness of the clones in the clusters. Knowing which clones are linking multiple clone classes and which clones are being represented by slightly varying line numbers provides a count of how

many unique clones are in the cluster. The more individually unique clones contained in a cluster, the more sections of code that cluster is related to and the more interesting the cluster is to evaluate.

4.4 Limitations

It should be emphasized that the clustering results are solely dependent on the clones that are initially detected. As seen in cluster SE-0, a separate sequence of conditionals was not clustered, because it was not detected as a clone. During the use of this process it must be emphasized that only clones are being considered and there is a possibility that the clustering will not contain sections of code that were not detected as clones. Also, depending on the total number of clusters generated, there may be situations where a few related clones are located in a different cluster, as seen in cluster MM-11.

Other observed shortcomings associated with the process described in this paper include the fact that the information retrieval process has been able to provide further grouping of clone classes, but the amount of data that must be analyzed is still the same. However, the groupings or clusters provide for the collective analysis of the related clone classes and the evaluation of the clusters can be prioritized to start with those clusters containing the higher similarity values. Without the process described in this paper, the additional relationships found from the clusters would not be available, in which case the programmer must manually determine whether certain clone classes are associated with each other. In addition, the observations of the clusters in Section 3 represent a small portion of the total clusters that were generated. The distinctive relationships of the clone classes in some of the clusters could not be defined clearly.

5. Related Work

The only known work that combines the fields of IR and clone detection was proposed in (Marcus and Maletic 2001), who used LSI on the comments and identifiers of the *whole* program to determine high-level concept clones, such as different implementations of a linked list. The detection is done by evaluating files that are strongly related to each other based on the clustering of documents (or sequence of lines of source code) that are associated to these files. The detection is based on the semantic relationships of the comments and identifiers through LSI. A related approach called *semantic clustering* uses LSI to find topics instead of clones (Kuhn et al. 2007). The corpus of this approach is also the comments and identifiers of the whole program. In both cases the entire source code must be evaluated after being divided into documents. Furthermore, no structural or syntactic information is considered in the detection of the clones (before the manual evaluation by the user). The process described in this paper incorporates a type of *filtering* by first performing clone detection on the source code and thus only sections of code that are known to be clones syntactically are considered in the LSI process.

Similar to the goal in this paper to group clones in order to provide better understanding of them, several classification methods have been proposed. Kapser and Godfrey provide a categorization based on the locations of clones in a clone pair (Kapser and Godfrey 2004). These clones can be in the same or different blocks, functions, files, and directories. As part of an effort to perform refactoring on clones, Balazinska et al. provide a classification based on differences

between clones (Balazinska et al. 1999). Single token differences include the types of global and local variables, parameters, and return value. Also, differences in a sequence of tokens corresponding to an expression or statement are considered. These differences are used to determine the potential opportunity for refactoring. Koni-N'Sapu classifies clone groups in an object-oriented program based on their respective locations in the class hierarchy (Koni-N'Sapu 2001). The clones can be in sibling classes, parent-child classes, classes in a common hierarchy, and classes in different hierarchies. These approaches utilize the syntax and structure of the clones to perform the classification. The relationships among the clones from these classification techniques are concrete, because the classifications are based on well-defined similarity properties. However, the classifying of clones is based upon stringent requirements of the predetermined syntax and structural similarity properties. Classifying the clones through a process that is less stringent and can yield additional relationships will complement the syntax- and structure-based classification techniques. Instead of focusing on the syntax or structure of the clones, our approach utilizes the identifiers of the clones to cluster the clones. Whereas current classifications provide concrete relationships among the clones, the clusters in our approach are not dependent on the well-defined structure of the clones and more dependent on the relationships of the identifier names between the clones.

The analysis of clones contained in the source code of a kernel is provided in (Antoniol et al. 2002). The clones of the Linux kernel were observed in multiple release versions to determine the evolution of clones. Livieri et al. also performed analysis of clones in multiple Linux kernel releases using a distributed version of CCFinder (Livieri et al. 2007b). A heat map of the coverage of clones among the different releases is produced. Our approach is the first to focus on the clones in one release of a different kernel which is the Windows NT kernel and performs analysis of the clones after clustering rather than between release versions.

6. Conclusion and Future Work

This paper introduces an information retrieval process in which a large number of clone classes (46K CLoC) in the Windows NT kernel source code are clustered using LSI to help determine relationships among the clone classes. The results of the process yield connections between clones in the clone classes that would not be detected by a clone detection tool alone. These connections range from variations in the syntax of the clones to the use of the clones in different contexts based on the code surrounding the clones. This information could assist a programmer to both understand how the clones are used and to assist when maintenance of the clones is required.

CCFinder uses the token-based representation of a program to perform its detection analysis. Other clone detection tools like CloneDR use a different representation, such as an abstract syntax tree (AST), which is parsed from the source code (Baxter et al. 1998). An advantage of using an AST representation is that the detected clones are more structured because they represent definite blocks in the code based on a given subtree of the AST. A future work is to use different clone detection tools to identify the clones. We have used CloneDR in previous work to introduce our clone visualization tool (Tairas et al. 2006). Some tools offer control settings related to the strictness criteria for detection. It would be interesting to perform a comparison with the results of a tool in which the most liberal detection configuration was selected (i.e., the least strict setting). Related to the presentation of the clusters, enhancing our clone visualization

tool to provide a visual representation of the clusters as a complement to the textual HTML reports may be beneficial during the analysis of the clusters.

In terms of the corpus that is used, including the words in the comments as possible terms may be considered in future clone clustering. With respect to terms, the observation of the most influential terms (or identifier names) in the generated clusters was done manually, which was part of the whole process of understanding the clones. Future work will consider statistically determining which identifier names are the most influential for each of the generated clusters. This should give greater insight into the relationship of the clones in the clusters. In addition, stemming the identifier to separate terms hidden in the identifier names will be done to further enhance the results of LSI.

The Microsoft NT kernel was used in the case study for this paper, which represents a commercial operating system. Commercial software are typically closed in the sense that only a select few programmers have access to the source code and provide the development expertise for the product. This is in contrast with open source projects where contributions from the public are generally welcome, which can swell the number of programmers involved in the project. A comparison of the level of cloning between open source and proprietary projects would be interesting to determine how much cloning exists and if unique characteristics of clones are evident. For example, a better understanding of the degree of clones and the clone types might provide some insight into how different development approaches lead to duplicated code. This is a future area of work that we plan to investigate.

For output data from applying the process described in this paper to the Windows NT kernel, please refer to the project web site (see Footnote 4).

Acknowledgements

We thank the anonymous reviewers who provided many helpful suggestions that assisted in improving the content and presentation of this paper.

This project is supported by National Science Foundation grant CPA-0702764.

References

Antoniol G, Villano U, Merlo E, Penta M (2002) Analyzing Cloning Evolution in the Linux Kernel. Information and Software Technology. 44(13): 755–765.

Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K (1999) Measuring Clone Based Re-engineering Opportunities. Proceedings of the International Software Metrics Symposium. Boca Raton, FL, 292–303.

Baxter I, Yahin A, Moura L, Sant'Anna M, Bier L (1998) Clone Detection using Abstract Syntax Trees. Proceedings of the International Conference on Software Maintenance. Bethesda, MD, 368–377.

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and Evaluation of Clone Detection Tools. IEEE Transactions on Software Engineering. 33(9): 577–591.

Collard M, Maletic J (2004) Document-Oriented Source Code Transformation using XML. Proceedings of the International Workshop on Software Evolution Transformation. Delft, The Netherlands, 11–14.

Deerwester S, Dumais S, Furnas G, Landauer T, Harshman R (1990) Indexing by Latent Semantic Analysis. Journal of the American Society for Information Science. 41(6): 391–407.

Han J, Kamber M (2006) Data Mining: Concepts and Techniques. San Fransisco: Morgan Kaufman, 2nd edition.

Jiang L, Misherghi G, Su Z, Glondu S (2007) DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. Proceedings of the International Conference on Software Engineering. Minneapolis, MN, 96–105.

Jiang Z, Hassan A (2007) A Framework for Studying Clones in Large Software Systems. Proceedings of the International Working Conference on Source Code Analysis and Manipulation. Paris, France, 203–212.

Kamiya T, Kusumoto S, Inoue K (2002) CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Transactions on Software Engineering. 28(7): 654–670.

Kapser C, Godfrey M (2004) Aiding Comprehension of Cloning Through Categorization. Proceedings of the International Workshop on Principles of Software Evolution. Kyoto, Japan, 85–94.

Koni-N'Sapu G (2001) A Scenario-Based Approach for Refactoring Duplicated Code in Object-Oriented Systems. Diploma Thesis. University of Bern, Bern, Switzerland.

Kuhn A, Ducasse S, Gîrba T (2007) Semantic Clustering: Identifying Topics in Source Code. Information and Software Technology. 49(3): 230–243.

Li Z, Lu S, Myagmar S, Zhou Y (2006) CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. IEEE Transactions on Software Engineering. 32(3): 176–192.

Livieri S, Higo Y, Matsushita M, Inoue K (2007a) Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder. Proceedings of the International Conference on Software Engineering. Minneapolis, MN, 106–115.

Livieri S, Higo Y, Matsushita M, Inoue K (2007b) Analysis of the Linux Kernel Evolution Using Code Clone Coverage. Proceedings of the International Workshop on Mining Software Repositories. Minneapolis, MN.

Marcus A, Maletic J (2001) Identification of High-Level Concept Clones in Source Code. Proceedings of the International Conference on Automated Software Engineering. San Diego, CA, 107–114.

Rieger M, Ducasse S (1998) Visual Detection of Duplicated Code. Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering. Brussels, Belgium, 75–76.

Rieger M, Ducasse S, Lanza M (2004) Insights into System-Wide Code Duplication. Proceedings of the Working Conference on Reverse Engineering. Delft, The Netherlands, 100–109.

Russinovich M, Solomon D (2005) Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000. Redmond: Microsoft.

Tairas R, Gray J, Baxter, I (2006) Visualization of Clone Detection Results. Proceedings of the OOPSLA Workshop on Eclipse Technology Exchange. Portland, OR, 50–54.

Zhao Y, Karypis G (2005) Topic-Driven Clustering for Document Datasets. Proceedings of the SIAM International Conference on Data Mining. Newport Beach, CA, 358–369.