Bachelor Thesis

Immanuel Wietreich

Selecting Attributes for Automated Clustering of Disassembled Code from Embedded Systems

February 14, 2014

supervised by:
Prof. Dr. Sibylle Schupp
Dipl. Ing. Arne Wichmann

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg
Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 14. Februar 2014

__________________________
Immanuel Wietreich
## Contents

1. Introduction 1

2. Background 3
   2.1. Reverse Engineering 3
      2.1.1. Embedded systems 3
      2.1.2. IDA pro disassembler 4
      2.1.3. IDA Script, IDA Python interface 5
   2.2. Cluster analysis in general 5
   2.3. Statistical Cluster analysis 7
      2.3.1. Partitional clustering 7
      2.3.2. Hierarchical clustering 9
   2.4. Formal concept analysis 11
      2.4.1. Mathematical aspects of formal concept analysis 12
      2.4.2. Analysis of concepts 14
   2.5. Tools 17
      2.5.1. Conexp 17
      2.5.2. In-Close 18
      2.5.3. Graphviz 18

3. Application to problem domain 19
   3.1. The available data 19
      3.1.1. Analyzing the data 20
      3.1.2. Data extraction 22
   3.2. The considered methods 24
      3.2.1. Applied formal concept analysis 27
      3.2.2. K-means test 28
   3.3. Truth finding 29

4. Evaluation 31
   4.1. Case study: Pistachio L4 micro kernel i686 31
   4.2. Data comparison 33

5. Future Work 37

6. Conclusion 39

A. Listings 41
1. Introduction

Throughout this thesis, we propose an approach to aid the reverse engineer of disassembled binaries during the program comprehension process by applying clustering techniques. Since formal concept analysis does not automatically differentiate between available attributes we need to preselect certain attributes for the clustering. From the possibly large set of attributes of disassembly code, we select attributes that exploit the property of shared memory usage in embedded system executables.

In this scenario the reverse engineer of disassembled binaries could not access the original source code of those compiled binaries. Disassembled binary is the machine code, which is converted into the assembler language for the reason of interpreting by humans. The disassembly lacks of information compared to the original code. It is a difficult task to reason about the purpose of the disassembly parts. The assembler code is produced by IDA pro disassembler. It has an assembler structure and consists in case of the considered embedded systems of thousands of lines of code. Each line of code is addressed by an unique identifier and further contains operators and operands. The disassembly is automatically separated into subroutines, which we also call functions. To extract information from the IDA disassembly environment the offered python interface is used.

In order to help to identify the purposes of the disassembled functions under the lack of knowledge about the context, the approach is to find similarities within the functions, such that we can build a cluster. A cluster is a formalized way to group objects with respect to their attributes. Grouping objects with a clustering algorithm means placing objects within a defined space and putting objects into the same cluster, if they are close together. What closeness means in each specific context varies from case to case. A distance metric needs to be provided. Using such a procedure we expect it is possible to state that functions in a cluster have something in common semantically. Commonalities or dependencies between functions are used to help to mine purposes of those. For sure the most critical part is the selection of meaningful attributes.

It is expected that especially in the case of optimized embedded systems there are connections between functions with certain properties. These properties can be used to form attributes, which can be compared, such that it is possible to find functions with common attributes. There are various, but limited possibilities to select attributes from the disassembly. E.g. there are two functions and a static memory address as a common attribute. The first function writes something to this address, the second one reads the content from the address. Thus it seems reasonable those two function are somehow related.

Within the thesis different possibilities of selecting attributes are discussed. Any kind of attribute combination will result in a different cluster. Therefore we are looking for the most meaningful selection. Furthermore there are also various types of clustering tech-
niques considered and compared. Statistical and semantic approaches are both covered. The most promising method is formal concept analysis, such that my work will focus on that method. The expected information gain by this or other clustering methods is a connection between functions, such that we can state that these functions belong together in some way. E.g. they do similar things, they have similar dependencies or use shared resources. Having this information, the reverse engineer can use it to avoid looking for connections manually.

So my work is about extracting data from the disassembly, selecting attributes from that data and choosing a clustering method to which we can apply the attributes. Therefore we have to compare all attribute combinations. An overview of the possible attributes, their combinations and influences is provided as well. The different methods we considered are compared. There are statistical and semantically methods. Both have advantages and disadvantages which we will compare within the thesis. It will be explained in detail how formal concept analysis works, why it is a good choice for us and how concepts can be interpreted in the context of our environment. Furthermore we also evaluate the clustering results against the truth, such that we can measure the relevance of the developed clusters.

The thesis consists of the following parts: In Chapter 2 (background) explains in detail how the problem state arises, which possibilities we have to encounter the challenge and give an overview of all tools that have been used. In Chapter 3 (application to problem domain) we will argue about why we chose formal concept analysis over other clustering techniques, which possibilities there are to select attributes for clustering and which kind of results can and cannot be expected. The implementation of scripts for extraction, parsers, filters and converters which led to the results are presented right after each explanation. The Evaluation and test of the results can be found in Chapter 4 (evaluation). In Chapter 5 (future work) we proclaim some further possibilities to encounter left issues. Finally in Chapter 6 (conclusion) we give a summary of the results and state in which way the goal was reached.
2. Background

In this chapter we introduce the operating context. In Section 2.1 reverse engineering of embedded using IDA is explained. Section 2.2 to 2.4 introduces statistical and semantic clustering methods. And in Section 2.5 there are some tools mentioned used to aid the process.

2.1. Reverse Engineering

Reverse engineering in general is about reconstruction of systems by examining structures, states and behaviors. It is an attempt to rebuild the original blueprint of the system. In the case of software engineering, reverse engineering is mostly about compiled software systems and the intention to extract the single elements and their relation. The compilation process as a one way function produces a binary file. This file contains machine code which can hardly be understood by humans. Compared to other fields of reverse engineering, in this case it can only be the goal to interpret the reverse engineered systems in some way, but there is actually no way to reconstruct the original source code entirely due to the information loss during the compilation process. The machine code consists of digit patterns which can be identified by a disassembler and converted into a more sophisticated representation.

There are some applications as reasons why we should do reverse engineering in the first place. Software testing is one reason. Even if a software system was tested on the source code level, it is not guaranteed that it also acts exactly as expected in its compiled binary form. Therefore the binaries can be reverse engineered and due to the disassembler tested not just like a black box. Every combination of input parameters and system states are interesting for possible resulting output. And this is the point for the other application. Bug fixing on binary level is necessary if the bug can not be identified on the source code level. This can either be an interpretation problem within the architecture dependent compilation process, or a system specific problem. In both cases reverse engineering will be the choice which leads to a statement.

2.1.1. Embedded systems

Embedded systems are computer systems within a specific technical context and usually with a very specific task like monitoring or controlling. One property of these systems is that they may have a small own operating system and a contextual program part. Embedded systems have only one purpose without any need to be changed on demand. Therefore they are closed and compact, such that they can be optimized along the provided task. This results especially in a more efficient and shared memory usage. General
systems have dynamic memory management, which usually means each process gets an
own memory section. Within an optimized embedded system, memory is more often di-
rectly addressed and shared by functionalities which do similar tasks or even calculations
which are correlated. So the total number of memory accesses is reduced. Due to that
fact, it is reasonable to say that functions which share the same memory might have
something to do with each other. There are several embedded system architectures like
PowerPC, ARM, MCS, Atmel, Coldfire and more.

2.1.2. IDA pro disassembler

The tool we use is the Interactive Disassembler (IDA). This tool is able to disassemble
binaries of all well-known hardware architectures (e.g. x86, PowerPC). The produced
assembler structure is row wise, one operator and one to three operands each. Every
row has its own hexadecimal address. Furthermore IDA separates the disassembly au-
tomatically into subroutines. Subroutines are blocks of code which belong together and
are executed sequentially. They can be called by other subroutines with parameters like
functions. Since they have the structure of programmed functions with input parameters
and return values, we can call the subroutines functions. The quality of the disassem-
bly depends on hardware architecture support. So the reconstruction of the code and
data structure can not be guaranteed. Disassembling is the first step of reverse engineer-
ing. The success is limited by the disassembler performance. Various properties may
change in different architecture types. They have different sets of operators, different
kinds of operands and also handle memory differently. This is why disassembling is more
complicated in rare architectures.

Figure 2.1 shows a small extraction of a disassembly. Sub_409990 is the generic label of
the function. As we can see by the hex address on the left, the actual line of code is the
same until "push ebx". The prologue sequence "push, mov, sub" is typical for x86 code.
There are to data references shown, ds:dword_40EF40 and ds:DeleteCriticalSection.

```
00409990: ----------------- SUBROUTINE ------------------
00409990
00409990 sub_409990 proc near ; DATA XREF: sub_409880+40F0
00409990 var_1C = dword ptr -1Ch
00409990
00409990 push ebx
00409991 mov eax, 3
00409996 sub esp, 1Ch
00409999 lock xchg eax, ds:dword_40EF40
0040999C cmp eax, 2
004099A0 jnz short loc_4099C3
004099A5 mov ebx, ds:DeleteCriticalSection
```

Figure 2.1.: Disassembly extraction from IDA pro
2.2. Cluster analysis in general

2.1.3. IDA Script, IDA Python interface

IDA itself does not support the export or extraction of the disassembly. But there is a command line interface within IDA provided. Additionally there is a plugin to extend the embedded CLI by a python compiler. With IDA Script and IDA Python it is possible to extract and export data from the disassembly.

**IDAPython** is an open-source general-purpose interpreted programming language. It is used because of the existing interface to the IDA pro disassembler. Due to the lack of interface support for the current version of python, here only version 2.7 is used. It fulfills the requirements to handle the extraction of data from IDA.

**IDAScript** makes it possible to extract disassembly data. To avoid text crawling, IDAScript is able to access the data structure and fields of the disassembly. Since the disassembly consists of different abstract parts, this helps to separate data from code, subroutines from other subroutines and to access every line of the disassembly by its hexadecimal memory address.

2.2. Cluster analysis in general

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar to each other than to those in other groups (clusters). To build those clusters there are two major concerns: deciding on position, size and border of the groups, and deciding on what similarity means for the specific application. There are various clustering techniques with different clustering algorithms which differ significantly in their notion of what constitutes a cluster and how to efficiently find them. Similarity measure within clustering is e.g. performed by distance measure, dense area of the data space, intervals or statistical distributions. So clustering can be formulated as multi-objective optimization problem or an iterative process of knowledge discovery.

**Requirements**

Clustering strongly depends on input data, parameters and the output expectations. Therefore it is necessary to analyze the data precisely and choose a clustering algorithm wisely. Converting and adjusting of input data to achieve meaningful output data is just as important as to clarify what could be stated by looking at the outcome. Since the outcome of a clustering technique is most of the times some numbers, it is important to make clear and to document which behavior shall be expressed by the outcome.
2. Background

Benefits and drawbacks

Clustering is capable to extract a lot of information out of huge amounts of data, where every manual attempt would take far too long. Characterizing or classifying objects with plenty of properties is a computationally intensive task, which can be accomplished by clustering. In reality data rarely fits the properties for a specific clustering technique. In order to adjust the data to match the requirements of an algorithm, some information might be lost during the transformation process, thus it is not always possible to invert the process. Additionally, there is never a guarantee that the result is what we expected. Due to approximation in some algorithms there is always just a chance to be right. In the end the result has to be interpreted in the right way. Not interpreting the result in the right way can lead to no or even a wrong answer. An example of a clustering result is shown in Figure 2.2. The elements in the space are grouped to either the blue, red or yellow cluster. Each element has certain properties which are responsible for their position in the vector space. In general close elements are grouped and named cluster. Therefore it is most likely these elements share some properties. That is the basic idea atop statements about the data are made.

Attribute influence

The attributes (or random variables in statistics) have the biggest influence on the entire clustering task. Based on the properties of the attributes the algorithms will produce results with different meaningfulness. Especially data correlation is a big effort. Correlation refers to any of a kind of statistical relationships including dependence. If there is a dependency between two data sets, one could be predicted by the other one. "Formally, dependence refers to any situation in which random variables do not satisfy a mathematical condition of probabilistic independence. There are several correlation coefficients, often denoted $p$ or $r$, measuring the degree of correlation. The most common of these is the Pearson correlation coefficient, which is sensitive only to a linear relationship between two variables (which may exist even if one is a nonlinear function of the other)" [2]. Usually in statistics a strong correlation between random variables is avoided. Ideally, the

\[ 1 \text{http://en.wikipedia.org/wiki/Cluster_analysis} \]
correlation coefficient for statistical applications is close to zero, such that probabilistic independence is reached. Therefore the random variables would be evaluated separately.

2.3. Statistical Cluster analysis

As mentioned before, there are several types of clustering procedures. In statistical cluster analysis we can distinguish the existing algorithms.

2.3.1. Partitional clustering

Partitional clustering is about partitions with certain properties you initially decide on. The size of the partitions can change during the procedure, but usually the basic idea is, that these partitions do not overlap. So for every element in the data space used for the analysis there has to be exactly one partition where this element belongs to. Furthermore the data space of such a clustering is a euclidean space, which means there is a common distance measure possible.

There are three major types of partitional clustering methods to be presented.

Centroid-based clustering

In centroid-based clustering, clusters are represented by a central vector. The number of centroids is predefined and element assignment is performed by distance measure between element positions and positions of the centroids within the vector space. A widely used algorithm for this kind of clustering is k-means. K-means clustering gives a formal definition as an optimization problem, such that solving this problem is NP-hard. Finding the best solution includes deciding on the number of clusters, the initial position of the clusters and the iterative process to adjust centroids until nothing changes. Therefore assigning members to the cluster is just a computational task, whereas choosing initial parameters is approximation. Figure 2.3 shows an example of a centroid based clustering. There are three centers with a different radius. Within the specified radius all elements belong to that cluster. The three lines in the graph are for visualizing the partitioning part of the space.

Distribution-based clustering

Distribution-based clustering is the closest method to statistics. It relies on distribution models. Like the Gaussian curve, there is a decreasing probability for elements to belong to that cluster. The highest probability is in the center of the cluster. Therefore the clusters do not need to be adjusted if new elements are put into the vector space. Even overlapping of probability areas of the clusters is not a problem, since for every element
there is a chance to belong to a specific cluster. As this technique works well in theory, it suffers from overfitting unless constraints are put on the model complexity. Overfitting occurs especially if the model is too complex, if there are too many parameters compared to the number of observations. The most used method to encounter this task is the expectation-maximization algorithm. It is an iterative method to find maximum likelihood or maximum a posteriori estimates of parameters in statistical methods. Figure 2.4 shows a distribution based cluster example. The elements are the same as in Figure 2.3, but the probability of each element belonging to one cluster decreases the higher the distance to the center of the cluster is. In the intersection of the probability space of the right two clusters can be seen that elements belong to the lower cluster, even if they are closer to the upper one. That is because the likelihood of the low cluster has a bigger radius factor.

Density-based clustering

Density-based clustering is about finding dense areas of data. Since dense is a relative declaration, comparisons overs the whole data have to be performed, under the purpose to detect areas of higher density. The most popular density based clustering method is DBSCAN. It provides a cluster method called "density-reachability". It connects elements within a certain distance threshold. A cluster consists of all density-connected objects (which can form a cluster of an arbitrary shape, in contrast to many other methods) plus all objects that are within these objects’ range. Another interesting property of DBSCAN is that its complexity is fairly low - it requires a linear number of range queries on the database - and that it will discover essentially the same results (it is deterministic for core and noise points, but not for border points) in each run, therefore there is no need to run it multiple times. The biggest problem arises while trying to separate nearby

1http://en.wikipedia.org/wiki/Cluster_analysis
2.3. Statistical Cluster analysis

clusters. If clusters are close together, the algorithm can not distinguish between them anymore. Figure 2.5 shows such a density based cluster. The cluster on the right side would never be a result of the other clustering types. A cluster of this shape is the result of the transitivity occurring in this method.

![Density based cluster example](http://en.wikipedia.org/wiki/Cluster_analysis)

**Figure 2.5.: Density based cluster example**

### 2.3.2. Hierarchical clustering

Connectivity-based clustering or hierarchical clustering creates a hierarchical decomposition of the set of elements using some criterion. There are generally two types of strategies: agglomerative (bottom up approach) and divisive (top down approach). In the agglomerative strategy every element starts in its own cluster and merges with other clusters while moving up the hierarchy. In the divisive approach all elements start in a single cluster, which is divided over some criterion as often until it can not further be split. The typical representation of such a hierarchical cluster is a dendogram. A dendogram visualizes not only the connections between objects and in which order they are merged, but also their distances shown by the height of the connection node. An example dendogram can be seen in Figure 2.7 and the corresponding raw data in Figure 2.6. Step by step close nodes are merged. The distance between nodes is kept by the length of the edges connecting the merged nodes.

There are further variants of clusters which may change some properties. For example subspace clustering is a possible approach if some dimensions of the data space are not relevant for a specific cluster, these can be reduced to have a more precise result within a subset. But in the end it is important to remember to keep the connection between the clusters upon the reduced dimensions. Furthermore there is a graph theoretical approach, which proceeds on the connectivity graph. Stronger connected nodes in that graph form a cluster, rather than less connected nodes, which means the edges are the indicator for the clustering process.

---

Fuzzy clustering

As another property of statistical clustering we distinguish between hard and soft clustering. The usual hard clustering puts every element in exactly one cluster. In the soft clustering, also called fuzzy clustering, every element corresponds to a cluster with a certain property. Applications are for example human gender (hard clustering) or hair color (soft clustering). Fuzzy clustering also implements the possibility for overlapping clusters. If there is an area between two clusters where these clusters overlap and the elements within this intersection belong with some property to both clusters, this is called an overlapping cluster.

No matter which method is chosen to cluster any kind of data. The limitations of all methods are bound to some kind of distance measure, which again depends on the dimension of the vector space. The properties which have to hold for distance measure are as follows.

Distance measure properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetry</td>
<td>$D(A, B) = D(B, A)$</td>
</tr>
<tr>
<td>Constancy of self–symmetry</td>
<td>$D(A, A) = 0$</td>
</tr>
<tr>
<td>Triangular Inequality</td>
<td>$D(A, B) \leq D(A, C) + D(C, B)$</td>
</tr>
</tbody>
</table>

Finally it is a non-trivial question to find the right method to apply to your data. Many properties have to be checked, adjustments have to be done and trade-offs have to be considered.

---

1http://en.wikipedia.org/wiki/Cluster_analysis
2.4. Formal concept analysis

Introduction  Formal concept analysis is a mathematical theory of the field of algebra. Basically it consists of binary relations which can be represented in a two-dimensional table. The entries of the table can be summed up to connected rectangles, if reordering is allowed. Simplified, the formal concept analysis is the theory of maximal rectangles in two-dimensional tables. It analyzes the algebraic properties of those rectangles and provides algorithms to calculate them.

Formalism  The elements of a maximal rectangle form two sets: objects $O$ and attributes $A$. Every maximal rectangle is also called concept, which consists of the components $O$ and $A$. Its representation is the tuple $(O, A)$. The terms used express some similarity of formal concepts with natural concepts. In natural concepts we can intuitively see when objects are related. In Figure 2.10 we can see that all fish live in water and the formal concept approves to that similarity. Additionally with formal concepts, there exists an order with an upper and lower concept. All concepts of a table build a context. In this context exists for every subset of concepts a distinct common lowest superset and a common biggest subset. The context of a relation illustrates plenty of its properties without interpreting or changing something. Usually experts perform the interpreting part of the concepts. Figure 2.8 shows how the maximal rectangles are built and represented in a lattice. On the left side there is the binary relation table of objects and attributes. In the middle the maximum rectangles are built around the binary relations. Now each of these rectangles corresponds to one of the concept nodes in the lattice on the right. The edges between the nodes in the lattice belong to the formalism of super-concepts and sub-concepts.

Figure 2.8.: example showing maximal rectangles for concept construction [9]
2. Background

Related work Applications of formal concept analysis in software engineering are most of the times about the component wise reuse of modules. Other fields of use are e.g. data mining, text mining, machine learning, knowledge management, semantic web and biology¹.

The implementation of formal concept analysis has two major tasks: first, the calculation of the set of all concepts, and since the context of the concepts is only implicitly given; it has to be calculated and provided by a data structure. Naive algorithms to calculate the concepts have exponential complexity. However, reality in the attempt to organize software components shows that it does not need to be expected. Therefore, there are different algorithms to perform that task.

To illustrate formal concept analysis, Figure 2.9 shows a binary relation table and Figure 2.10 the corresponding concept lattice. For example, using reordering of columns, owl and bird share the attributes bird, predator and air and form a rectangle. This is represented as the lower left concept in the lattice. Furthermore Figure 2.10 shows not only concepts as nodes and super-concept/sub-concept relations as edges, but also the contained objects by white boxes and attributes by gray boxes. The color of the upper half of a concept node is dark if an attribute is introduced. The same behavior yields for the lower half and objects.

<table>
<thead>
<tr>
<th>dog</th>
<th>mammal</th>
<th>bird</th>
<th>predator</th>
<th>gregarious</th>
<th>water</th>
<th>land</th>
<th>air</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>cat</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>owl</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hawk</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>budgie</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>whale</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>dolphin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shark</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tuna</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wolf</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>frog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 2.9.: Animal example to illustrate formal concepts

2.4.1. Mathematical aspects of formal concept analysis

Formal concept analysis starts with the formal context, which is the basis of the analysis. The formal context is a binary relation. [9]

¹http://en.wikipedia.org/wiki/Formal_concept_analysis
Definition 1 (Context): A formal context is a triple \((O, A, R)\) consisting of the following sets: objects \(O\), attributes \(A\) and the relations \(R \subseteq O \times A\) between them. One relation can be written as \((o, a) \in R\), which means object \(o\) has attribute \(a\). A context defines in which relation a set of objects bear to a set of attributes. A formal context is mostly represented as a context table. There rows are objects, columns are attributes and the crosses represent the relation between them.

Definition 2 (common objects/attributes): For a set \(A \subseteq A\) of a context \((O, A, R)\) the common objects are defined as \(A' = \{o \in O | (o, a) \in R, \forall a \in A\}\) and the common attributes for the set \(O \subseteq O : O' = \{a \in A | (o, a) \in R, \forall o \in O\}\).

Definition 3 (concept): A concept of a context \((O, A, R)\) is a pair \((O, A)\) \(\in 2^O \times 2^A\) where it holds: \(O' = A\) and \(A' = O\). The set of all concepts of a context is denoted as \(C(O, A, R)\). \(O\) is considered to be the extent of the concept and \(A\) is the intent of the concept.

Theorem 1: If \((O, A, R)\) is a context and if \(O, O_1, O_2 \subseteq O\) are sets of objects and \(A, A_1, A_2 \subseteq A\) are sets of attributes, it holds:
1. \(O_1 \subseteq O_2 \implies O'_2 \subseteq O'_1\)
2. \(O \subseteq O''\)
3. \(O' = O''\)
4. \(O \subseteq A' \iff A \subseteq O' \iff O \times A \subseteq R\)
Same properties for sets of attributes.
Definition 4: If \( c_1 = (O_1, A_1) \) and \( c_2 = (O_2, A_2) \) are concepts of the context \((O, A, R)\), then \( c_1 \) is subconcept of \( c_2 \), if \( O_1 \subseteq O_2 \). This is formalized by \( c_1 \leq c_2 \). A subconcept contains fewer objects than its superconcept, whereas the objects of the subconcept share more attributes, such that: \((O_1, A_1) \leq (O_2, A_2) \implies A_1 \supseteq A_2\).

Concept lattice: Every formal context \((O, A, R)\) contains a set \( C(O, A, R) \) of formal concepts with a complete partial order. The set of all concepts form a complete lattice, the concept lattice.

Definition 5 (lattice): An ordered set \( V = (V, \leq) \) is a complete lattice, if for every subset \( V \subseteq V \) the supremum \( \bigvee V \) and the infimum \( \bigwedge V \) exists.

For every set \( C_S \subseteq C(O, A, R) \) of concepts of a context exists a unique smallest superconcept \( \bigwedge C \) and a largest subconcept \( \bigvee C \).

The Hasse diagram (like Figure 2.10) of the lattice visualizes every vortex as a concept and every edge as a \( \leq \)-relation. For a better overview the Hasse diagram does not only visualize the subconcept-superconcept-relation, but also due to the transitivity of the \( \leq \)-operation visualizes all indirect reachable superconcepts (or subconcepts). A Hasse diagram is shown in Figure 2.10.

2.4.2. Analysis of concepts

This section presents data structures and algorithms to calculate concepts of a context and their lattice. They are the requirements to perform the concept analysis (in software engineering). There were various algorithm developed over the years, but we will only have a look at the naive approach, Ganters original algorithm and the currently most efficient algorithm In-Close. Even if we are not interested in manipulating existing algorithms, it is important to compare them for a simple reason. Computing concepts is a combinatorial task and therefore depends on the number of possible permutations. Since we are only interested in a small fraction of the permutations, it is necessary to reduce the overhead. Formal concept analysis quickly gets computationally intensive, so this is a major aspect.

The concepts of a context: Every formal context has a non empty set of concepts. Their maximum size grows exponentially with the size of the context. A Context \((O, A, R)\) with \(|O| = m\) and \(|A| = n\) has maximum \(2^l\) concepts, where \( l = \min(m, n) \).

Naive algorithm: A naive method to calculate all concepts of a context \((O, A, R)\) is given by using theorem 1: every pair \((O', O)\) with \( O \subseteq O \) is a concept, because \((O')' = O'\) yields. Analogous to that, every pair \((A', A)\) with \( A \subseteq A \) is also a concept. If \(|A| \leq |O|\) yields, then \( \{(A', A^O)|A \subseteq A\} = C(O, A, R) \) is the set of all concepts and the naive algorithm calculates \(2^{|A|}\) pairs. The disadvantage of this method is obvious: no matter what amount of concepts are in \( C(O, A, R) \), the time complexity is \(O(2^n)\) with \( n = \min(|O|, |A|)\). The operations ‘ and ” have the time complexity \(O(\min(|O|, |A|))\).
Therefore the naive algorithm has a final complexity of $O(2^n \times |O| \times |A|)$ with $n = \min(|O|, |A|)$. Thus it is not appropriate for real applications.

**Ganters algorithm:** The algorithm of Ganter calculates the set of all concepts of a context with a much better time complexity $O(|O|^2 \times |A| \times |C(O, A, R)|)$ as the naive algorithm. Therefore a total order of concepts is defined and beginning with the smallest concept in this order the respective successor is determined. Every step generates from the last found concept up to $|O|$ candidates. So the next concept will be found after maximal $|O|$ steps, or no further concepts exist. Because of that the complexity of this algorithm is limited by the true amount of concepts and $|O|$. Ganter defines a total lexical order $\prec$ for sets which are built on total order of the universal sets $O$ and $A$: $O = o_1, ..., o_O$ with $o_1 \prec o_2 \prec ...$, analog for $A$. [9]

Nowadays Ganters previously well performing algorithm is outdated.

**In-Close algorithm:** The algorithm with the best known performance is the In-Close algorithm. The general method is to add attributes, one at a time, to a current intent. As each new attribute is added, the corresponding extent is computed. This is achieved by intersecting the current intent $A$, with the attribute extent $\{j\}'$, of the new attribute. I.e. $\text{new } A = A \cap \{j\}'$. The new extent can be one of three outcomes [1]:

1. The empty set.
2. A non-empty set smaller than $A$.
3. The same set, i.e. $A \cap \{j\}' = A$.

In-Close uses the lexicographic approach for implicitly searching, but avoids the overheads of computing repeated closures [1]. Instead, it completes closure incrementally, and only once per concept, as it iterates across the attributes, using outcome 3 as an indicator to add the new attribute to the current intent and prune the recursion at that point. Iteration across the attributes continues, adding a new attribute and pruning its branch each time outcome 3 occurs, until the concept is closed. Another argument for the implemented In-Close algorithm within the command line tool is the possibility to define the minimum intent and extent for the resulting concepts. This is not only a useful filter if we are looking for bigger concepts, but also reduces the number of computations and therefore provides more efficiency again.

Explanation of In-Close in Figure 2.11 [1]:

**Line 3** - Begin a new concept.
**Line 4** - Iterate over attributes, starting at attribute $y$.
**Lines 5 to 8** - Form an extent, $A[r_{new}]$, by computing $A[r] \cap \{j\}'$.
**Line 9** - If the extent is empty (outcome 1), skip the recursion and move on to the next attribute. Note that a value $< 0$ can be used if concepts with extents below a certain size are not of interest.
**Lines 10 and 11** - If the extent is unchanged (outcome 2) then add the attribute to $B[r]$ (incremental closure), skip the recursion and move on to the next attribute.
**Lines 12 and 13** - Otherwise, the extent must be a smaller, non-empty intersection
2. Background

1 InClose(r, y)
2 begin
3 \( r_{new} \leftarrow r_{new} + 1; \)
4 for \( j \leftarrow y \) upto \( n - 1 \) do
5 \( A[r_{new}] \leftarrow \emptyset; \)
6 foreach \( i \) in \( A[r] \) do
7 if \( I[i][j] \) then
8 \( A[r_{new}] \leftarrow A[r_{new}] \cup \{i\}; \)
9 if \( |A[r_{new}]| > 0 \) then
10 if \( |A[r_{new}]| = |A[r]| \) then
11 \( B[r] \leftarrow B[r] \cup \{j\}; \)
12 end
13 if \( \text{IsCanonical}(r, j - 1) \) then
14 \( B[r_{new}] \leftarrow B[r] \cup \{j\}; \)
15 InClose(r_{new}, j + 1);
16 end

Figure 2.11.: In-Close algorithm [1]

(outcome 3), so call IsCanonical to see if it has already been generated.

Lines 14 and 15 - If the extent is not found, the concept is canonical, so initialize \( B[r_{new}] \) and call InClose to begin its closure. Otherwise, if the extent is found, skip the recursion and move on to the next attribute.

Explanation of IsCanonical in Figure 2.12 [1]:
The procedure searches for \( A[r_{new}] \) in the blocks of columns in-between those in B[r].

Line 4 - Take each column in \( B[r] \) (in reverse order) and
Line 5 - iterate down to that column from a starting column.

Lines 6 and 7 - In each column, try to match the extent. i.e. \( A[r_{new}] \subseteq \{j\} \)?

Line 8 - If the extent is found, stop searching and return false (it is not canonical).

Line 9 - Skip the column just iterated down to, ready to iterate the next block of columns.

Lines 10 - 13 - Finally (if not already found), search for the extent in the block of columns down to 0.

Construction of the lattice structure: Any concept calculating algorithm produces the set of concepts of a context, which hold a lattice structure only implicitly. To build a lattice it is necessary to know which concepts the neighbors are. The structure gets explicit using a Hasse-diagram: it shows concepts as vertices of an undirected graph and the \( \leq \)-relation between two concepts as an edge between vertices of the graph. For application of formal concept analysis the lattice structure needs to be available like the data structure of a Hasse diagram. A concept lattice as a data structure does not only contain concepts of a context, but also establishes relations between them. A concept in a concept lattice could contain the following information:
2.5. Tools

In this section tools used for this thesis are introduced. They would not have been mandatory for evaluating results like IDA pro, but they chosen to support the process of calculating and visualizing information. Since the data we work on is complex, visualizing helps to understand the content.

2.5.1. Conexp

Conexp\(^1\) is a Java based interactive concept browser. The tool is not only capable of loading, calculating and visualizing concepts from a cxt files, conexp establishes also an interactive concept graph. The produced lattice graph is live rendered to be interactive, which is useful until the graph grows to fast. Within the lattice it is possible to move

\(^1\)http://conexp.sourceforge.net

---

2.5. Tools

IsCanonical \(r, y\)

Result: Returns false if \(A[r_{new}]\) is found, true if not found

begin
for \(k \leftarrow |B[r]| - 1\) downto 0 do
  for \(j \leftarrow y\) downto \(B[r][k] + 1\) do
    for \(h \leftarrow 0\) upto \(|A[r_{new}]| - 1\) do
      if not \(I[A[r_{new}][h]][j]\) then break;
      if \(h = |A[r_{new}]|\) then return false;
    
y \leftarrow \(B[r][k] - 1\);
  for \(j \leftarrow y\) downto 0 do
    for \(h \leftarrow 0\) upto \(|A[r_{new}]| - 1\) do
      if not \(I[A[r_{new}][h]][j]\) then break;
      if \(h = |A[r_{new}]|\) then return false;
  return true;
end

Figure 2.12.: IsCanonical algorithm [1]

- Object and attribute set.
- Links to direct super and sub concepts.
- A set of all objects which are not contained in object sets of subconcepts and a set of all attributes which are not contained in attribute sets of superconcepts.

Depending on the application there could be any other information contained in a concept.
the concepts for a better manual exploration. They are represented as circles with an upper and lower half. Depending on the color of the half, there are introduced objects or attributes, which means they are not inherited. Additionally all objects and attributes of the concepts (extent and intent) can be shown around the concept node itself, or as context menu. An example lattice produced by conexp is shown by Figure 2.10.

2.5.2. In-Close

In-Close\(^1\) is a command line tool to calculate concepts from cxt files. In matters of performance, this is the most efficient tool to calculate concepts we currently know. It implements a more efficient concept calculating algorithm than the one of Ganter. It is programmed in C and named the In-Close algorithm. Simon Andrews, the developer of this algorithms states it is on average more than 1300 times faster than Ganters algorithm \[1\]. The specified features from the website are:

- 1 million concepts per second mining
- context reduction
- Burmeister .cxt input/output

Execution parameters are minimum intent (least amount of attributes per concept) and minimum extent (least amount of objects per concept). Therefore it is possible to filter concepts while creation. The output is a reordered cxt file (such that tools like conexp can read them faster) and a concept file, in which every row corresponds to one concept.

2.5.3. Graphviz

Graphviz\(^2\) is a powerful open source graph visualization software. Graph visualization is used to build diagrams of structured information. Diagrams help to understand that structure. Graphviz provides lots of options and supports different visualization strategies, which is the reason we chose this tool. We use Graphviz to visualize concept lattices in a static way and for evaluation reasons also for connectivity graphs, such as it is easier to verify the results. The special format Graphviz needs is parsed directly by the python extraction tool or could also be produced by converting the cxt file. An example graph created by Graphviz is shown by Figure A.4.

\(^1\)http://inclose.sourceforge.net
\(^2\)http://www.graphviz.org
3. Application to problem domain

As the goal is to find a way to use clustering upon the disassembly to extract related functions, such that it might be easier for the expert, the reverse engineer, to identify the purpose of these functions, the main steps are as follows.

To decide whether clustering leads us to the goal, we first have to think about available clustering methods. And for comparing different methods against each other, we need to know what data we can use. Thus the first step is to acquire data we can use for clustering.

3.1. The available data

As we know from Section 2, we can get the entire disassembly separated as functions, consisting of a sequence of operators and operands. The set of operators and the set of operand types is fixed and depends on the hardware architecture. The amount of different operators can exceed hundred and the number of different operand types is usually below twenty. As we would like to compare functions, it seems to be a good approach to keep that structure while extracting that information from the disassembly. Treat functions as objects to be compared is the first step. The functions in the disassembly are not necessarily ordered, such that we can not argue about closeness of functions within the disassembly.

Thus it is necessary to find other kinds of properties of the functions we can measure and compare. The next approach deals with the operators and operands. Comparing the finite set of operators we have some possibilities. We can build a vector with n elements with \( n = \text{number of operators} \). For each function the vector is filled with the counts of the occurring operators. All vectors together form a relation matrix, which is our first possible data for clustering. As the occurrence of single operators may not characterize the functions in a sufficient way, we may want to know about occurrences of combinations of operators. Thus we select distinct pairwise operators as tuples, and even triples. Of course the accumulated sequence of operators could be increased until the full length of operators in a function is reached, but doing that is way to specific for clustering. The combinations of operators will further be called singles, doubles and triples.

After taking operators into account, we get a deeper look on the operands. As we realize quickly, not every operand provides usable information. Since we are looking for operands which can be used to establish connections between functions, we only want a specific type of operand: memory references and any other kind of memory address. The other operand types, no matter if static or dynamic, can not provide any connection. Usable memory addresses can be found in operand types 2 to 7 (see Table 3.1). After we are able
3. Application to problem domain

Table 3.1.: Assembler operand types

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>bad operand number passed</td>
</tr>
<tr>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>General Register (al,ax,es,ds...)</td>
</tr>
<tr>
<td>2</td>
<td>Memory Reference</td>
</tr>
<tr>
<td>3</td>
<td>Base + Index</td>
</tr>
<tr>
<td>4</td>
<td>Base + Index + Displacement</td>
</tr>
<tr>
<td>5</td>
<td>Immediate</td>
</tr>
<tr>
<td>6</td>
<td>Immediate Far Address</td>
</tr>
<tr>
<td>7</td>
<td>Immediate Near Address</td>
</tr>
<tr>
<td>8</td>
<td>FPP register</td>
</tr>
<tr>
<td>9</td>
<td>386 control register</td>
</tr>
<tr>
<td>10</td>
<td>386 debug register</td>
</tr>
<tr>
<td>11</td>
<td>386 trace register</td>
</tr>
<tr>
<td>12</td>
<td>Condition (for Z80)</td>
</tr>
<tr>
<td>13</td>
<td>bit (8051)</td>
</tr>
<tr>
<td>14</td>
<td>bitnot (8051)</td>
</tr>
</tbody>
</table>

to select for specific types, we can extract them easily. We distinguish between memory references (type 2), base+index (type 3 and 4) and immediate addresses (type 5-7). The reason why these memory addresses are helpful appears especially in embedded systems. They are built to be compact and to optimally use the available memory. Thus in embedded systems there is a lot inter-functional memory usage. This can be interpreted as a connection between functions.

A last and not yet covered possible connection between functions is the most obvious one. Functions call other functions. This is in software analysis often called a "caller-callee-relation" and used for any kind of inter-module-relationship [11, 5, 10]. Using that relation, we take every function not only as object, but also as possible attribute of them. Treating function calls in the same way, we may also count occurrences.

3.1.1. Analyzing the data

Now there are several kinds of possible attributes for our objects to use for clustering: three kinds of attributes taken from operators, three kinds of attributes taken from operands and the function calls. Having this data, we need to argue about which clustering method could be applied. To do that, we put all the data in a big table and examine for properties. A property which is independent from the method is the correlation of the data. Several ways to calculate a correlation coefficient exist in statistics, but we actually do not have typical random variables. A random variable is a variable whose value is subject to variations due to chance, we have occurrences. So the correlation of the extracted disassembly data has to be expressed in a more semantic way. First of all we are not interested in the degree of correlation, so we take the data as binary relation.
3.1. The available data

In this context correlation means two or more functions have at least one attribute in common. The more objects share the same attributes and vice versa, the stronger is the correlation in general. But the usability of that correlation depends on the density. So strong correlation is good if and only if it is limited to a number of columns and rows, such that the correlated subtable is less correlated to other subtables or even independent. In statistics independence means that the occurrence of one does not affect the probability of the other. Beside the non-applicable probability, this is what we can also find here.

Looking at Table 3.2 objects 1-4 are correlated, since 1 shared an attribute with 2 and 2 shares an attribute with 3 and so on. Thus the entire data is correlated, but the use is very little. Trying to group the objects due to that table is useless. We can directly see that without applying any method.

\[
\begin{array}{cccc}
\text{A} & \text{B} & \text{C} & \text{D} \\
1 & X & X & \\
2 & X & X & \\
3 & X & X & \\
4 & X & & \\
\end{array}
\]

Table 3.2.: Correlation example 1

Another example (Table 3.3) shows a much more meaningful table. There is no correlation between the upper left and the lower right quarter. Thus we would intuitively group objects 1-2, and 3-4. Furthermore we can see, that the density within one of those quarters is very high, which makes it more likely these objects have something in common.

\[
\begin{array}{cccc}
\text{A} & \text{B} & \text{C} & \text{D} \\
1 & X & X & \\
2 & X & X & \\
3 & & X & X \\
4 & & X & X \\
\end{array}
\]

Table 3.3.: Correlation example 2

To relate these hypothetical aspects with real data, it has to be clarified that we are looking at potential substructure, rather than complete tables. With those two examples the two extrema, which may never occur exactly like that, are shown. It is only an expression of reason why the results may look like they do.

The next thought is about the attribute counts. As mentioned it is possible to count occurrences within the functions, but it is not clear yet how to handle them. Let them be values of the relation table would results in the loss of possibility of the binary relation table representation. The other possibility would be adding every combination taken
3. Application to problem domain

into account into the list of attributes, such that we have attributes like "three times mov". Since no data is reasonably analyzable with a lot more attributes than objects, we disregard this idea. In the end we leave it to the method if the counting is relevant or not.

3.1.2. Data extraction

The first step of the implementation is the data extraction. A binary file is required and a disassembler like IDA. As described in Section 2 there is a python plug-in for IDA, with which it is possible to execute python scripts directly on the disassembly. Also necessary for data extraction is the IDA script language to access the disassembly fields. Some functions used to access the data structure are shown in Figure 3.1.

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segments()</td>
<td>returns a list of all segments (code,data,...)</td>
</tr>
<tr>
<td>Heads(seg_ea, SegEnd(seg_ea))</td>
<td>returns all addresses within a certain range</td>
</tr>
<tr>
<td>GetMnem(head)</td>
<td>gets instruction mnemonics (operator)</td>
</tr>
<tr>
<td>GetFunctionName(head)</td>
<td>gets subroutine/function name</td>
</tr>
<tr>
<td>GetOpType(head, i)</td>
<td>gets type of the operand nr. i</td>
</tr>
<tr>
<td>GetOpnd(head,i)</td>
<td>gets name of operand nr. i</td>
</tr>
<tr>
<td>GetOperandValue(head, i)</td>
<td>gets value of operand nr. i</td>
</tr>
</tbody>
</table>

Figure 3.1.: Some IDA script functions used to reach the data

For large binaries the extraction process can take quite long, because within the extraction phase every line of code has to be touched. Therefore it is necessary to take care of the memory and decide on which attributes shall be selected in the beginning. For the general and variable use of the extraction tool two major execution parameters can be set. In the first place there are the attribute types for which the extraction tool is crawling for. A list of possible types is shown by Figure 3.2. A "1" enables so search for that type, a "0" disables it. In the case of the operand types it is possible to look for type 2-7, which are the different relevant cross reference memory types. The runtime complexity of the algorithm is linear, but it is limited to the amount of available memory. For larger applications the necessary memory can grow fast.

After having collected the data from the disassembly there are options for attribute combinations. As Figure 3.3 shows, every element appended to the combination list results in a single cxt-file. Therefore the first combination added in this example would result in callers and all considered operand types, the second would include all considered operator types.

Furthermore the combination of function connectivity (caller-callee) and the memory references can be converted into the dot-file format. Graphically represented by Graphviz, the connectivity between functions can be easily verified by the expert this way. This
3.1. The available data

```python
types = {}
types['singles'] = 1
types['doubles'] = 1
types['triples'] = 1
types['mixed'] = 0
types['callers'] = 1
types['optype'] = '234567'
```

Figure 3.2.: Extraction type options within python tool

```python
combinations = []
combinations.append(['callers', '2', '34', '567'])
combinations.append(['singles', 'doubles', 'triples'])
```

Figure 3.3.: Attribute combination options within python tool

representation shell provide a better overview than any build-in connectivity graph. An example output built from L4 pistachios pingpong binary is shown by Figure A.4.

The entire python scripts can be found in the appendix. Upon the main script "ida_extraction.py" there are further scripts written in the context of this thesis.

- **cxt_merge.py** - Merges two cxt files with the same objects.
- **concepts.py** - Using pure python FCA version of Dominik Endres. It is about 6x slower than in C and build on C. Lindigs Fast Concept Analysis work. Even if it is slower, it provides a lot more customization, as it acts like a library.
- **concept_statistics.py** - Creates a statistic based on intent/extent of In-Close output.
- **kmeans_test.py** - Partitional clustering experiments, more in Section 4.2.

**The dot format** is used to formalize software visualization graphs and e.g. used by Graphviz. There are numerous options to change the way any kind of data structure can be visualized. Its content format equals appendix Figure A.1.

**The cxt format** is a short representation of binary relations as a table with column and row names. This format is used by about every tool dealing with formal concept analysis. There exists another file format with the extension 'cxt', so do not get confused. The cxt files we deal with start a "B", followed by the numbers of objects and attributes, then line by line the corresponding names and in the end the relation matrix with symbols "X" and ".". An example file is shown by Figure A.2.
3. Application to problem domain

3.2. The considered methods

After knowing about the data, we have a look at clustering methods. They have different properties which we know from Section 2.

Statistical clustering maps the data into a multidimensional space. Every attribute of the data results in another dimension and every object is put into that space. Especially for partitional clustering the curse of dimensionality has a big impact. Partitional clustering is based on euclidean distance measure:

\[
d(a, b) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}, \text{ for } a, b \in \mathbb{R}^n
\]

For the improbable case of a rich filled high dimensional space this distance measure gets equidistant. For an example space of 500 dimensions, every dimension has a little influence on the entire sum of the formula. As in Section 4 will be demonstrated, the malfunction of the distance measure under these circumstances will lead into a very few big clusters and a lot empty clusters, which is not what we were looking for. For any other and more expected case the high dimensional space is almost empty. If there are almost no objects in the space, compared to the size of the space and even the few which there are, building a cluster is just as useless.

Considering the data, it is likely that some attributes are more important or have more influence than others. Thus we can try to reduce the dimensions by eliminating less relevant attributes. Semantically reducing attributes may only work if we know about the influence in the context. Since we do not know anything about the context in the first place, because this is a naive and automated approach, we have to focus on automated dimension reduction.

Latent semantic indexing (LSI) for example can be used to fulfill that task. The underlying mathematical concept is the singular value decomposition (SVD)[17, 15]. Without going into detail with that concept, the basic idea is to perform some matrix multiplications of orthogonal basis and eigenvector calculations, such that depending on eigenvalue selection a certain amount of dimensions remains. On the first view this is great, but dimension reduction is always connected to information loss or accuracy loss.

Precision is something we really want. Knowing that two methods are just maybe connected is not enough. And dropping some attributes just to embellish the results of the algorithm in acceptance of information loss is not reasonable in the first place. It could be useful in a different scenario, but we are most interested in getting a complete overview of connected functions. No matter if we decide to use density-based, centroid-based or distribution-based partitional clustering, the curse of dimensionality remains.

On the other hand there is the hierarchical clustering with the two major approaches bottom-up and top-down. Those two approaches are not that different at all, only in their runtime complexity. The top-down approach is difficult, because we need to decide in every step of the hierarchical tree on the next attribute to separate for. One issue is
3.2. The considered methods

the size of the possibly produced clusters, the corresponding subtrees in that structure. Due to the attribute distribution over the space, there would be very big ones and very small ones. The issue with even more impact is again the correlation. The aim of the hierarchical clustering is to separate the tree more and more, such that one subtree fulfills a certain attribute and the other subtree does not. Having objects which do not fit into one subtree or the other, we need to separate on a higher level, such that we need to add another branch. That makes the hierarchical decision tree non-binary, flat and therefore inappropriate. This is shown by Figure 3.4. Object 1 and 4 are distinct, but 2 and 3 cannot be distributed to either 1 or 4.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.4.: Hierarchical correlation example

The bottom-up hierarchical approach suffers from the same property, even if the procedure is a different one. Every object begins with its own cluster. Then the cluster merges with other clusters containing similar attributes. It can also be explained using example Figure 3.4. The resulting merged clusters would look like Figure 3.5. In the next step all clusters would be merged to one big cluster. So the information gain is very poor, since the goal is to produce high intra-dependence and low inter-dependence. Therefore hierarchical clustering would also not be the ideal choice.

Figure 3.5.: Bottom-up approach example
Figure 3.6.: Concept lattice negative example 1 - low inter and intra correlation

Semantic method: formal concept analysis

After considering different statistical methods which might be applied to our disassembly data, we compare those against a semantic approach. Formal concept analysis offers another way to handle such data.

The first question to answer is about the comparability. Statistical and semantical methods have fundamental differences. Statistical methods rely on approximations, probabilities and reduction of probably less important information, whereas semantic methods are precise, computationally intensive, which results mainly from the basic idea to work on words instead of just numbers. Furthermore, formal concept analysis does not reduce anything, but establishes groups based on binary relations. This makes those two approaches not directly comparable, but in order to match the requirements of the problem, it is enough. Both, statistical and semantical approaches, are clustering methods and therefore meet the requirements as a solution for the problem.

Formal concept analysis is the choice we pursue. As it is a sharp procedure, it provides us not only the possibility to know which other objects are in the same cluster (concept), but also exactly which attributes those objects have in common. This is a very important step for reverse engineering, since we need to know as much as possible about which attributes are responsible for the connection between the objects. Even if the clustering worked out well, the expert still need to be able to interpret the resulting concepts. Automated interpretation of the results and automated proof of correctness is, if it is even possible, not covered within this thesis. It is a non-deterministic task to check the results against the original source code in the case it is available. If it is not available, there is no way to be sure about the results. Therefore the expert would never be replaced, just supported.

To be a little more specific about what our result should look like, we compare some examples. In example 1 (Figure 3.6) the inter-correlation is small, which is actually good, but the concepts and therefore the intra-correlation are too small. Since this example does not provides any information gain at all, it is the worst case. Fortunately this case only occurs when attributes are not selected carefully.

In example 2 (Figure 3.7) a quite meaningful context is shown. In the absence of a real metric to measure what a useful context and its corresponding lattice should look like,
it can only be semantically explained. Even if every concept can be taken as a cluster, we want them to have decent distance in matter of connectivity and a small amount of links. The deeper the concept in the lattice and the more connections it has to other concepts, the more attributes are shared by different concepts. This is an optimal but surrealistic example.

Example 3 (Figure A.3) shows a high inter-correlation. Due to that issue the intra-correlation is blurred and can not really be expressed. Even if there are plenty different concepts, the expectation to a usable context needs to result in separate paths between top and bottom element as good as possible. Since this example is most reasonable, it has to be analyzed carefully. Even with the given properties there are useful informations, but most likely only in some concepts.

![Concept lattice useful example - low inter and high intra correlation](image)

### 3.2.1. Applied formal concept analysis

There is no formal concept calculation algorithm developed within this thesis. The existing implementations of In-Close and pure python by Lindig suffice to encounter the problem. For larger binary files In-Close should be used due to the performance issue. The pure python implementation has an advantage as well. Since python scripts are live compiled, it is easy to change the implementation on demand and therefore the way the input data is handled and especially how the output concepts look like. A major benefit of the python implementation in order to build a lattice are the calculated neighbors. Having the neighbors per concept, constructing the lattice is just connecting concepts without any further calculation. Additionally, being in position to vary the output, it is also possible to perform an individual ordering right away. So concepts with a higher interestingness could be put atop. What interestingness means can also vary from case to case. Basically a relatively big amount of combined objects and attributes in one concept is preferred over just a big amount of objects or attributes. Additionally, corresponding to the correlation, a relatively low amount of neighbors is preferred as a secondary criterion.

In the attempt to implement an appropriate visualization for constructed lattices, the first idea was using Graphviz and the dot format. Graphviz has a great algorithm for
ordering and sorting connected nodes. But the concept nodes in this case may need a bunch of labels for objects and attributes to identify them. Using this static graph visualization does not match the special requirements. Either some labels need to be pruned, or the nodes grow large to cover all names of objects and attributes. So even if it is a neat visualization possibility, it lacks of interactivity.

Conexp as one of very few interactive concept explorers meets the requirements with some deductions. As it is a java application, it suffers from the limitations of the virtual machine. More than a few hundred concepts can not be visualized in a useful way. But within these limitations exploring concepts like described in previous sections is an appropriate way to verify concept lattice structures.

3.2.2. K-means test

A statistical method have been tested, even though we figured out in before that they would not suffice to match our requirements. The K-means clustering, representing the partitional clustering, was chosen to proof the argument. The K-means implementation used is from the python library plug-in scipy. Modules vq (vector quantization), k-means and whiten were used. The vq module is a toolkit to handle vectors in general. The whiten module is used to reduce the Gaussian noise of the test data. K-means was applied to various kinds of selected data, even to data with occurrence counts, which has no influence on formal concept analysis. All test data sets have in common that the amount of attributes is bigger than 100 and in some cases even bigger than the amount of objects. For example in the case of an extended connectivity data set, there are all objects also represented as attributes plus some memory references.

No matter which data set was applied to the k-means algorithm, the result was always the same. Increasing the initial amount of clusters iteratively resulted automatically in a lot of empty clusters. Choosing 5, 10 or 20 clusters, all objects were only separated into 2 clusters. Every other cluster was empty. Therefore the curse of dimensionality [8] has a huge influence. The proclaimed problem really occurred. Trying to reduce dimensions[4] is not an option in this context. It would force us to rate the importance of single attributes, which is not possible, since this is a zero knowledge system.
3.3. Truth finding

In order to verify the results of an analysis usually a comparison with well known data is performed. Generally, there are two possibilities for verification. Either the experts knowledge is taken or the specific comparison of the analysis results and the original data source. The verification we call truth finding, because we are interested in the fact how close we are to the truth, the correctness of our clustering result. In reverse engineering the truth finding is more complicated. No analysis result would represent exactly the structure or purpose of the original source code, due to the information loss during the compilation process (Figure 3.8). The analysis results can only be compared semantically with the original code. In case of the function calls it is more likely to find those in the original source code, requiring them to be explicitly used. For the case study in Section 4.1 some of those comparisons are done and in fact provide verification about the success of clustering correctly. Most likely impossible to verify are connections associated to memory references, except there are explicit memory accesses performed within the source code.

Figure 3.8.: Abstract truth finding - analysis results compared with source code

C is the language with which most of the binaries are compiled from. So compared to other programming languages, in C it is possible to address memory directly. That makes C most likely the best performing language.

To cover and validate the entire process from source code functions to concepts, a short C program is written. The source code is shown by Figure A.7. Beside the main function, there are two functions (add, mult) called by the main function and global variable (test_global). Figure A.8 shows an extraction of the corresponding call graph, which
visualizes the dependency between these functions and the memory in the context of the entire compiled binary. The shown user code is represented in only five elements within this call graph with actually 83 elements (69 functions, 14 memory references). All other elements belong to library code which is also compiled and put into the binary file.

Figure 3.9.: C program test concept

Calculating the formal context from the C program, there are 58 concepts. One of them is shown by Figure 3.9 and represents our user code of the program.
4. Evaluation

In this chapter results of applying the data to the methods is discussed. The differences between attribute combinations and the corresponding outcomes will be shown. Furthermore a detailed case study on the Pistachio micro kernel\(^1\) compares all considered attribute combinations and in the data comparison different binaries and their results are compared.

4.1. Case study: Pistachio L4 micro kernel i686

To show the occurring differences between various attributes and to measure the success a case study on the Pistachio L4 kernel is performed. These binaries are chosen to represent the properties we can detect upon those kinds. This kernel is suitable because its size fits for testing, the original source code is provided and the disassembly binaries are fully labeled. Therefore it is easily possible to verify the correctness of resulting concepts.

Since the number of objects is fixed for all kinds of attribute combinations, we do not have to mention it for every case. The number of objects is 306, the total lines of code are 21408. First of all there are the seven kinds of attributes we picked for testing. Further combinations of them are possible without limitations.

The different attribute types, their numbers and the resulting number of concepts is shown by Table 4.1. In this example no base memory addressing (optype 3,4) exists, so this attribute can be ignored. Before combining attributes we first need to know what they offer solely.

Before building concepts, we would like to have an overview of the data we have extracted. The six graphs in Figure A.5 try to visualize that for every attribute kind. They show

\[\text{http://www.l4ka.org/65.php}\]

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>#attributes</th>
<th>#concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>optype 2</td>
<td>operand type 2</td>
<td>186</td>
<td>80</td>
</tr>
<tr>
<td>optype 34</td>
<td>operand type 3,4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>optype 567</td>
<td>operand type 5,6,7</td>
<td>3309</td>
<td>5002</td>
</tr>
<tr>
<td>callers</td>
<td>functions calling other functions</td>
<td>274</td>
<td>182</td>
</tr>
<tr>
<td>singles</td>
<td>distinct single operators</td>
<td>84</td>
<td>29927</td>
</tr>
<tr>
<td>doubles</td>
<td>tuples of operators</td>
<td>719</td>
<td>1367413</td>
</tr>
<tr>
<td>triples</td>
<td>triples of operators</td>
<td>2634</td>
<td>475136</td>
</tr>
</tbody>
</table>

Table 4.1.: Attribute overview
accumulated attribute occurrences. For every attribute it is counted in how many objects it occurs and due to that distributed to a class. In the first graph the value \((2,29)\) means in this context there are 29 different attributes which have independently 2 objects in common. Therefore it is an indicator for potential concepts. In every case the amount of attributes decreases for increasing amount of common objects, as expected. That means either it is a small combination of attributes and therefore the possibility to occur more often is a priori higher or the combination occurs in more objects because its relevance is higher.

We can learn something from those graphs. Most of the attributes have only a few objects in common and only some of them have a larger amount of objects in common. This is helpful to know, because there are two properties we do not want: attributes occurring in only one object (which are filtered before generating the graphs) and attributes which seem to occur in a major subset of objects. The information gain would be very little and could not be used. What we also can say about the graphs is, that attributes on the left side are more individual ones and on the right side there are more widely used attributes, probably system functions or application wide memory buffers.

Performing the concept calculation, Figure 4.1 shows the resulting number of concepts. As we can see, the number of concepts does not only depend on the number of attributes, but on the correlation of the binary relations. To get a quick overview of the resulting concepts of our six types, especially about the sizes of them, Figure A.6 shows a distribution of concepts depending on their intent and extent sizes. For each of the six graphs the horizontal axis is the size of the intent, the vertical axis is the size of the extent. If there are multiple concepts with the same intent and extent sizes, they are summed up and visualized by their color. The colors are blue (few members) up to red (many members). Analyzing these graphs, the scaling has to be considered. The concepts with only one object in the extent have already been pruned.

Looking at the concept distributions graphs we can again learn something. Singles, doubles and triples have similar distributions of concepts. The highest density of concepts is between 5 to 20 attributes and 5 to 50 objects. So the majority of concepts using operators have more than 5 objects in the extent, which is not reasonable. It is improbable to have so many functions with similar functionalities or dependencies. The upper tail and the right tail might be worth of further inspection, but only interesting in rare cases. Operand types 2 and 567 also seem to have similar distributions. Operand types 567 seem to have mainly memory references which do occur in groups, rather than alone. The sparse looking graph of operand type 2 tells us there are mainly smaller concepts with less than 5 objects or attributes. Actually that is quite helpful, since precision about correlated functions is what we want. Smaller concepts relate functions more directly.

The caller concept graph, not considering the outlier, is straight forward. No function calls more than 5 other functions and except some outliers no function is called by more than 10 functions.

To get more precise about the content and the value of the resulting concepts, one concept is picked to be shown. Figure 4.1 shows a call-graph with various queue-functions.
4.2. Data comparison

In comparison of a list of binaries (shown by Table 4.2) we can state that the file size, the number of lines of code and the number of functions only give a small hint on each other. In general larger files also contain more functions with more lines of code. This depends on the data part in the binary and the length of the functions. The analyzed binaries are either compiled for x86, 6811 or PowerPC. So these binaries from Table 4.2 we would like to compare from the perspective of concepts with different attributes.

<table>
<thead>
<tr>
<th>File name</th>
<th>File size</th>
<th>Arch.</th>
<th>LoC</th>
<th>#functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>pistachio-ia32-0.4 i686-kernel</td>
<td>167 KB</td>
<td>x86</td>
<td>21408</td>
<td>306</td>
</tr>
<tr>
<td>pistachio-ia32-0.4 i586-kernel</td>
<td>159 KB</td>
<td>x86</td>
<td>21452</td>
<td>306</td>
</tr>
<tr>
<td>In-Close.exe</td>
<td>145 KB</td>
<td>x86</td>
<td>6161</td>
<td>168</td>
</tr>
<tr>
<td>test.exe</td>
<td>50 KB</td>
<td>x86</td>
<td>6506</td>
<td>116</td>
</tr>
<tr>
<td>schiffeversenken</td>
<td>4 KB</td>
<td>x86</td>
<td>344</td>
<td>21</td>
</tr>
<tr>
<td>embedded1</td>
<td>52 KB</td>
<td>6811</td>
<td>6389</td>
<td>112</td>
</tr>
<tr>
<td>embedded2</td>
<td>527 KB</td>
<td>ppc</td>
<td>114820</td>
<td>2177</td>
</tr>
<tr>
<td>unins000.exe</td>
<td>833 KB</td>
<td>x86</td>
<td>166074</td>
<td>3597</td>
</tr>
<tr>
<td>vlc.exe</td>
<td>124 KB</td>
<td>x86</td>
<td>10425</td>
<td>146</td>
</tr>
<tr>
<td>gvedit.exe</td>
<td>163 KB</td>
<td>x86</td>
<td>19927</td>
<td>1144</td>
</tr>
</tbody>
</table>

Table 4.2.: binary test file overview

Table 4.3 lists the binaries with the corresponding attribute types each row. The numbers
are resulting concept counts. As we can see the concepts from the caller or optype 2 attributes are quite small compared the three operator attributes. The optype group 5, 6 and 7 varies strongly between different binaries. Optype 3 and 4 seem to be used only by a few binaries. Even optype 2 has an outlier and is not used by embedded2. The two binaries of size greater than 500 KB and attributes optype 567, doubles and triples resulted in an amount of computations to calculate the concepts that even the In-Close concept miner crashed.

<table>
<thead>
<tr>
<th>File name</th>
<th>Caller</th>
<th>Type 2</th>
<th>Type 34</th>
<th>Type 567</th>
<th>Singles</th>
<th>Doubles</th>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>i686-kernel</td>
<td>182</td>
<td>80</td>
<td>0</td>
<td>5002</td>
<td>29929</td>
<td>1367413</td>
<td>475136</td>
</tr>
<tr>
<td>i586-kernel</td>
<td>380</td>
<td>77</td>
<td>0</td>
<td>9394</td>
<td>16767</td>
<td>1583811</td>
<td>466249</td>
</tr>
<tr>
<td>In-Close.exe</td>
<td>110</td>
<td>161</td>
<td>0</td>
<td>461</td>
<td>1565</td>
<td>32197</td>
<td>14449</td>
</tr>
<tr>
<td>test.exe</td>
<td>103</td>
<td>58</td>
<td>0</td>
<td>529</td>
<td>2172</td>
<td>38426</td>
<td>8893</td>
</tr>
<tr>
<td>schiffeversken</td>
<td>29</td>
<td>68</td>
<td>0</td>
<td>91</td>
<td>167</td>
<td>855</td>
<td>371</td>
</tr>
<tr>
<td>embedded1</td>
<td>62</td>
<td>124</td>
<td>289</td>
<td>761</td>
<td>42695</td>
<td>732551</td>
<td>276221</td>
</tr>
<tr>
<td>embedded2</td>
<td>1588</td>
<td>0</td>
<td>717</td>
<td>err</td>
<td>405639</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>unins000.exe</td>
<td>9408</td>
<td>1169</td>
<td>0</td>
<td>err</td>
<td>74278</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>vlc.exe</td>
<td>172</td>
<td>88</td>
<td>0</td>
<td>822</td>
<td>1880</td>
<td>39534</td>
<td>25912</td>
</tr>
<tr>
<td>gvedit.exe</td>
<td>246</td>
<td>621</td>
<td>0</td>
<td>1255</td>
<td>1349</td>
<td>46829</td>
<td>197073</td>
</tr>
</tbody>
</table>

Table 4.3.: Concepts per file per attribute
The information we can extract from the Table 4.3 is mainly the fact that caller and optypes seem to produce less concepts and therefore as expected relatively more reasonable ones. Having over half a million concepts, even if some of them might be useful, is not helpful for the reverse engineer as long as there is no real metric for the interestingness of a concept. Of course the amount of concepts can increase if attribute types are combined, more than just the summed amounts.

We know from the case study that the ratio between objects and attributes does matter. The amount of objects increases inversely proportional to the amount of attributes. In the x86 code e.g. printf is a function which is called by almost every other function, but itself only calls one function. This would be represented by a concept with a lot of objects, but only one attribute. A main or initiator function would look contrary. This would be a function which calls a lot of other functions but is never called by any of them. Therefore the concept would contain one object and many attributes. So even if the thought is obvious that larger concepts provide more reasons to state the commonality of objects, the border cases give also a big hint on the program structure.

Considering these border cases, we can for example separate user code and library code more easily. The library code used to be called only, such that it is a candidate to occur only as attribute in combination with object code of the user code. If it is possible to isolate the cases where the library code calls other library functions, this task can be accomplished quite well.

**Result of the analysis.** Concepts are potentially meaningful, but due to the lack of a real interestingness metric they can not be rated automatically. Therefore an expert is still needed to interpret and rate the resulting concepts. A general best performing attribute combination can not be stated over all binaries. Especially the use of memory references is very different comparing hardware architectures and embedded systems. Therefore which memory references are taken to calculate concepts needs to be considered in every case. Generally a combination of function calls and memory references performed best in the test.

The reverse engineer has limited tools which may aid him within the exploration process. The best build-in tool in IDA is the call-graph. In the expectation the reverse engineer has some knowledge about the binaries and therefore what might be more interesting and expressive, these concepts give a real hint for functional dependencies and common tasks. All in all the concepts accelerate the process of function purpose discovery.

As we have seen the quantitative evaluation of the resulting clusters is not completely satisfiable. The qualitative consideration proclaims a very good use of some concepts, a good use of lots of concepts, but also a weak use of some remaining concepts. In a real application it used to be most important to find some significant candidates to get started with the program comprehension. Therefore finding the very good concepts which tell a lot about the program is a huge step. Afterwards they can be excluded from further discovery to grant more overview.
Refering to the pistachio case study we analyzed occurrences of member functions of the scheduler class. The basic attributes in this case were callers. It appeared that different member functions from type scheduler and prio_queue share same concepts. To be precise there were 15 concepts out of 174 containing one of both. This proofs the complex data structures is consistent and can be found again in the disassembly.

Something which has to be considered during the concept exploration can be understood by looking at the concept lattice. The more attributes an object does have, the bigger is the chance to share a subset of the attributes with other objects. So objects might be part of many concepts, an entire sublattice. Collecting the concepts of the sublattice borders the impact range of the considered function. Especially in the beginning of the program comprehension of a completely unlabeled disassembly there are only hints like this where to start looking for meaningful concepts.

In the end of the evaluation we can state that the goals were reached in a sufficient way. The attempt to aid in reverse engineering might not be the best, but any aid is a progression. Supporting the function discovery was not the only goal in this thesis. We found out a lot of facts about what we could and what we could not do with the disassembly. Clustering definitely is an appropriate approach for this task. The main problem remains in verification of the results, but this is more a problem of reverse engineering, rather than the procedure. We also found out a lot of further possibilities which are covered in the future work chapter.
5. Future Work

Even though we decided in our approach to use mainly an attribute combination of function calls and various memory references with formal concept analysis, there are still some promising ideas. One approach is about using inter-connectivity and intra-connectivity within the connectivity graph (caller-callee-graph) [11, 10, 5]. So no transformation into data which fits for some clustering technique needs to be performed. The goal of this technique is to group participants by minimizing inter-connectivity and maximizing intra-connectivity. By adding memory references to the connectivity-graph as only called participants this feature could be covered. The resulting groups might not be deterministic and different optimal results are possible. The advantage would be a result set of clusters which is compared to formal concepts relatively low.

An optimization step for the current procedure is about stack offsets. Memory references which belong to the stack often do have offsets and other memory references have relative addresses. As long as the memory is not allocated dynamically, it should be possible to calculate absolute memory addresses or stack positions. The advantage of doing this could be the equalization of reference labels which actually do refer to the same address, but due to their different start address and offset it is not clear right away. For example one pointer points to "3+1" and the other to "2+2" which is actually the same address. This means there is potentially a lot more common memory usage which is just not discovered yet. But in the end there is a risk. Calculating absolute addresses using offsets or just dropping the offset could both lead to inconsistency.

Another possibility which could be interesting belongs to the idea analyzing the operators. Pushing the stacking of operators as attributes not just until three, but until the entire operator sequence of a function is covered, might be a complete different approach. Since there would not be any intersection between functions with this kind of attribute anymore, nothing else but direct function duplications could be found. The idea is more about calculating $tf \times idf$ values like in information retrieval that every function obtains a vector containing these weight values. Upon those vectors a cosine similarity measure can be performed, which will provide information about functions being close to each other in the vector space. The statements made on this basis are for sure different compared to formal concept analysis, but potentially interesting. One disadvantage is about the runtime. Time complexity, due to the fact we have to perform the cosine similarity to every pair of two functions, will be $(n-1)!$.

$sim(d_i, d_j)$ for $[i \neq j|i, j \in F], F = set \text{ of all function}$

The last idea about a way to improve the results is the use of semi-supervised learning instead of unsupervised learning. Semi-supervised learning is expected to generate more sophisticated results due to some knowledge we provide in advance. If this technique really performs better depends strongly on the possibilities to train a classifier. There-
fore training data and verification data is needed. The issue in the context of reverse engineering is the difference between binaries. For every kind of binary a separate classifier has to be trained. Kinds of binaries are in this case the architecture type, the exact compiler, embedded system or not and possibly more. So training suitable classifiers might get complicated, but afterwards this technique should be more sophisticated.
6. Conclusion

In this thesis, we showed how formal concept analysis could be used to cluster disassembled code. Considering some constraints, the initial goal has been reached.

The first question on the topic of clustering disassembled functions could have been answered quickly. The individual extraction of the disassembly data, using the IDAScript command line interface together with the python plug-in, allows to catch every information the disassembler is able to reconstruct from the binaries. Therefore the basis for selecting attributes is given. The developed python tool which does the extraction of data also converts the data directly into the necessary form to use it for clustering.

For the purpose of clustering the main difference between embedded and non-embedded systems is about stack usage. Embedded systems are not encapsulated and do not need to share the hardware with other applications. So they have unlimited access to all resources, which we use to enhance the range of possible attributes. But even if the stronger connectivity of functions due to the memory references in embedded systems may enhance the results, it depends completely on the quality of the disassembly.

After evaluating seven general types of attributes, the analysis indicates some attribute combinations are more reasonable than others. Especially the combination of function calls as one attribute combined with one or more shared memory reference types achieved helpful results. The analysis shows variably represented occurrences of memory reference types over different types of hardware architectures. Therefore this has to be considered first while performing the analysis.

In order to compare various methods processing the data we mined using the given attributes, some relevant differences were found. As expected formal concept analysis was preferred over statistical clustering techniques. The information loss while reducing dimensions in statistical methods is only applicable for the need of a general statement about the data. However, formal concept analysis as a sharp procedure performs well ordering the data in a way which is helpful for the reverse engineer.

It has been shown for some experimental cases that the resulting concepts match the truth, the original source code from which the binary was compiled. Therefore it is a proof by example for the correctness of the concepts. Nevertheless, the interpretation of these concepts remains for the reverse engineer.
A. Listings

```plaintext
graphtype graphname {
    node [option=value]; # change appearance of every node
    edge [option=value]; # change appearance of every edge
    ...
    node1 [options]  # node definitions
    node2 [options]
    ...
    node1 -> node2 [options] # node connections
    ...
}
```

Figure A.1.: dot file example showing how a dot file is constructed.

```plaintext
B

number of rows
number of columns
row1
row2
...
column1
column2
...
..XX..X.X
X...X...X.
..X.....X.
X.XXX.XXX
```

Figure A.2.: cxt example shows how a cxt file is constructed.
Figure A.3.: Concept lattice example 3, realistic but small - high inter and low intra correlation
Figure A.4.: Connectivity graph example from L4 pistachios pingpong binary. In parentheses are in-links and out-links. Labels of edges are the number of calls and the text labels are the corresponding function names.
Figure A.5.: Attribute/Objects stats
Figure A.6.: Concept stats
#include <stdio.h>

int test_global = 10; // define global var

struct test_struct {
    char *name;
    int len;
};

int add ( int a, int b ) {
    int sum = a + b + test_global; // read global var
    test_global += 1; // write global var
    return sum;
}

int mult ( int a, int b) {
    int ret = a * b + test_global; // read global var
    test_global -= 2; // write global var
    return ret;
}

int main( int argc, char* args[] ) {
    struct test_struct names[test_global];
    int peng = 0;
    for(int i=test_global;i>0;i--) {
        names[i].name = "name";
        names[i].len = add(i,i);
        peng = mult(i,i);
        printf("%s - %d - %d\n",names[i].name,names[i].len,peng);
    }

    return 0;
}

Figure A.7.: C program for testing
Figure A.8.: Extraction of call graph from C program for testing
Bibliography


