TUHH
Hamburg University of Technology

Bachelor Thesis

Leif Willerts

Clone Detection For Reverse Engineering of Disassembled Code

July 31, 2014

supervised by:
Prof. Dr. Sybille Schupp
Dipl.-Ing. Arne Wichmann

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

TUHH STS
Software
Technology
Systems

Eidesstattliche Erklärung

Ich, Leif Willerts, versichere an Eides statt, dass ich die vorliegende Bachelorarbeit mit dem Titel „Klonerkennung für Reverse Engineering von Assemblercode“ 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 31. Juli 2014

Leif Willerts

Contents

1 Introduction

Clone detection has been developed for a long time mainly to manage source code repositories and eliminate unnecessary duplication. Increasingly, other use cases for algorithms that try to find similar code fragments have been realized, including some outside of software development, engineering and management — especially in the field of reverse engineering. The reconstruction of concepts and models of functionality that underlie the software implemented by others is close to the nature of clone detection, in that sophisticated clone search algorithms also need to abstract away from some of the specifics of a piece of code which depend more on its context than on its intrinsic "meaning" in order to find other fragments of code which at least roughly share its functionality.

The challenges that reverse engineers dealing with disassembled machine code have to face are founded in its lack of abstraction mechanisms, such as control flow constructs, types, data structures or even classes, and encapsulation. A large part of assembly code deals with implementation details that a source code programmer would never have to worry about, using a compiler that constructs the executable machine code for him. Especially the fact that the deep tree-like syntax that makes encoding relationships between individual tokens of source code possible is not present invalidates many approaches that clone detection researchers have taken before. Also, identifying functions whose functionality is only related and which might use different values and refer to different data and code locations to fulfill a similar role inside a program is not trivial and requires processing code at a level of individual instructions and tokens.

In addition to that, code written on such a low level as assembly code is necessarily idiomatic, it does not only repeat itself whenever the same mechanism of a higher programming language is used, but also implements many different small tasks using the same instruction codes. Conversely, the use of general-purpose registers of various types will occur in almost any place in code, irrespective of context, since memory locations are often not addressed directly (either by necessity or to improve performance especially upon repeated use of the value retrieved from memory), so without considering the surrounding operations and what values could possibly occupy a register at the time of executing an instruction using it as an operand, the information that precisely that register is used is without any merit.

While it is hard to empirically measure the effectiveness of a clone detection algorithm in the absence of a prior definition of what constitutes a "clone" and what kind of code regions should be considered similar and reported, the usefulness in a particular known setting can be evaluated. This thesis tries to assess if the detection of code clones is a helpful and efficient strategy in reverse engineering, that is if supposed links between the functionality of a function (found only by capturing part of the instruction content itself, without higher-level dependency analyses) help associating each function with a definite functionality and therefore purpose in the program. A large embedded system, including the corresponding bootloader, consisting in total of over 2000 functions is processed based on an algorithm originally suggested in the literature for improving the quality of

software, and potential weaknesses of the approach and the implementation are discussed both in theory and in conjunction with the findings of evaluating the detection results.

2 Related Work

2.1 Reverse Engineering

Reverse engineering is a general term describing any activity that transforms a more concrete (low-level) into a more abstract (high-level) representation of a human-made product [1], or a complete system into separate parts, which in turn improves comprehension of the system and leads to a high-level description of its function based on (more simple) knowledge about the parts. For example, taking apart electronic devices to learn about their design and the components and then possibly repair them can be construed as reverse engineering, as could be even reading source code to understand the author's intentions [2]. Reverse engineering has gained a high degree of importance within the information-based economy that is the software industry. Obtaining all information necessary to produce a piece of software is equivalent to obtaining the software itself; there are no manufacturing costs and no additional knowledge or expertise is required in order to reproduce the artifact.

The explicit use of reverse engineering, that is, not within the normal workflow of writing and reading the representations currently worked on, can appear in many different contexts, some but not all of which are integrated into the process of again "forward-engineering" a new, modified or fixed piece of software. Achieving compatibility with not well documented systems and interfaces can be made easier by examining the given system to find out how to access its features, bugs can be found manually, custom features can be added, and the structures and specific algorithms can be imitated by another developer to create similar software. Testing and maintenance activities also require extensive knowledge of the system's architecture and its intended behavior [2]. The objective of reverse engineering a system can also include an assessment of its general trustworthiness and authenticity (see Section 2.3.1), or information needed to obtain data encrypted by algorithms implemented inside the analyzed code.

In the context of this thesis, which was motivated by a concrete scenario in which comparing different versions of a software system represented in assembly code (see Section 2.2) is instrumental in understanding the purpose of their components, version control mechanisms are also regarded as a form of reverse engineering. While technically the differences between the versions are merely reported, there is something to be learned about the functionality of the versions (especially one relying on knowledge about the other) from the results of such a comparison. In fact, it is intrinsically connected to gaining higher-level information about the system because a collection of found differences is not a working system and meaningless without the context they occur in.

2.2 Machine and Assembly Code

The application system examined in this thesis is already written in a language to be executed directly by a processor and is therefore specific to its architecture. Such lan-

guages are called “machine” or sometimes “native” languages or code, where “assembly” languages are the mostly equivalent languages that present the contents of executables, libraries and objects in a more human-readable format than simply being written down numerically (usually in base 16) as machine code. These languages are necessarily very close to the actual execution of single instructions cycle-by-cycle in a processing unit, and therefore lack many of the useful abstractions found in higher-level programming languages most software is written in today.

They need to specify every small operation that is to be performed, and the usage of parameters, control and data structures is made explicit in fine detail, as no assumptions can be made about the structures and features of possible source languages used. A machine is supposed to accommodate the use of any number of programming languages and paradigms, as well as different compilers, and provides the capability to execute simple steps from which complex programs can be constructed. While more complex instructions may be introduced to concisely represent common higher-level concepts in hardware and facilitate programming directly in assembly, these are generally not used frequently by many compilers (which are responsible for most of the machine code produced today, as opposed to humans writing assembly). They are often relatively domain-specific or were introduced at one time and later became less relevant but still present for backwards compatibility, and often they are impairing performance because the effort involved in decoding these instructions or the memory access times are more significant than the saved instruction cycles. Examples for instructions that are lengthy or complex to execute are `PUSHAD` and `POPAD`, which push and pop all general-purpose registers to and from the stack, respectively, as well as the mnemonics `SCASW` and `SCASD`, which share the same opcode by which they are represented in binary code. These mnemonics compare a memory operand with a register value of a size depending on the operand, set status flags as usual after a comparison, and then increment/decrement (in accordance with the direction flag) the source and destination index by the same size in bytes [3]. Also, assembly and machine code differ widely from architecture to architecture, due to different address and data bus sizes, register and cache dimensions, pipelining models, calling conventions, stack frame layouts, memory segmentation, threading, and simply the total class of operations the processor is capable of executing (the instruction set). Sections 2.2.1 and 2.2.3 below present several examples of what assembly languages typically look like, taken from or inspired by various documents describing the architecture [4–6].

Disassembly By default, a computer does not make the effort of storing native executables and libraries in a more spacious, readable text format by themselves, but since the machine language and the corresponding assembler language are strongly related, there are programs that convert one to the other: assemblers encode assembly code in binary form and disassemblers produce assembly code from it. The result of the execution of a disassembler (called “disassembly”) and can be easily read by an engineer versed in the corresponding language and provides a basis for interpreting the program (*not* in the sense of an interpreter, which executes source code instead of compiled code, but in the sense of discovering its functionality), editing and analyzing it further. A disassembly

can be parameterized, corrected and annotated by a reverse engineer to make it as close as possible to the likely assembly that the original program could be compiled to, and to increase its understandability yet without attempting a de-compilation into a possible high-level language. The disassembler may have wrongly interpreted sections of binary data as instructions or vice versa, wrongly formatted data in terms of sizes and string formats, or not recognized functions and their boundaries correctly, and arrays and complex data structures have to be made explicit by the engineer, optionally function and constant names added manually can make the interpretation of the code easier. Also, the capability to add comments in addition to meaningful names to a disassembly file is provided in many disassembly tools.

A popular interactive disassembler is IDAPro, originally authored by Ilfak Guilfanov and now developed, distributed and supported by HexRays SA [7]. Disassembly produced using IDAPro and already edited by a reverse engineer was used in this thesis. Properties of assembly code in general, the platform developed on in particular, and the software system subject to analysis, but also some that arise from correcting a given disassembly and adding information and structure to it by hand, will be discussed in the following to be considered in the design of a clone detection algorithm.

2.2.1 Subroutines

While other ways of structuring machine code in memory exist, like segments or sections, as well as possibly interleaving bytes, the disassembler should be able to abstract or ignore them, and the only functional unit larger than the instruction in common machine or assembly languages is the subroutine. Following popular usage and for ease of understanding and convenience, I will refer to them as functions even though they rarely satisfy the definition of a mathematical function: A stateless entity that produces definite results independent of anything but their input and not cause side effects which influence the computation of others.

Assembly code is written from top to bottom, where that direction is equivalent to increasing memory addresses where the instructions are stored and, inside functions, the order in which they are executed by continuously incrementing the instruction pointer. Applying control flow analysis to functions (requiring knowledge of which instructions "jump", i.e. modify the instruction pointer register to point to somewhere else than the sequentially following instruction) produces a subdivision of functions in basic blocks as exemplified in Figure 2.1, within which instructions are guaranteed to be executed in order from top to bottom. The `loc_...` markers on the left are added by the disassembler to mark the locations that some branch or jump instruction refers to, and these locations, as well as those instructions themselves, delimit basic blocks (symbolized by a dashed line in the figure). Arrows from the conditional branches in this code example point to their respective target location. Commas separate the operands of each instruction, and "C-style" comments are used in the Figures of this thesis.

On the one hand, assembly code is highly idiomatic, because certain common tasks have to be performed over and over again that either implement common patterns in the original source code, or deal with the necessary management of stack frames and

```
             clrlwi r5, r4, 24
loc_0000004: --------------------
     ^       lbz r4, 0(r3)
     |       addi r3, r3, 1
     |       cmpw CR0, r4, r5
     |   /-- bne lt, loc_0000004   // may jump to the address below (conditional)
     |   |   --------------------  //
     |   |   addi r3, r3, -1
     |   V   b lt, lr
loc_000001C: --------------------
     |       cmpwi CR0, r4, 0
     \------ bne lt, loc_000001C   // may jump to the address above (loop)
             --------------------  //
             li r3, 0
             b lt, lr
```

Figure 2.1: A library function (`strchr`) consisting of multiple basic blocks

registers. Not only the first and last instructions in many functions are the same, but higher level code constructs used frequently can cause the assembly to contain numerous corresponding duplications as well: For example `switch`-statements in a C-like language will likely be translated by a compiler into machine code with an instructions sequence similar to the one in Figure 2.2 — and even inside that code there are some instructions are repeated again and again.

```
             lwz r3, x             // load from x (memory address) into r3
             cmpwi CR0, r3, 1      // if x is 1
             beq CR0, loc_0000014  // enter the first branch
             cmpwi CR1, r3, 2      // if x is 2
             beq CR1, loc_0000020  // enter the second branch
             cmpwi CR2, r3, 3      // if x is 3
             beq CR2, loc_000002C  // enter the third branch
loc_0000014: --------------------
             [...]
loc_0000020: --------------------
             [...]
loc_000002C: --------------------
             [...]
```

Figure 2.2: A typical implementation of a three-way `switch`-statement

On the other hand, there is the possibility that compilers which eagerly optimize code for a given platform, and make their decisions depending on the context, cause generally idiomatic sequences of instructions to differ slightly from one place in the code to another.

```
                              // x,y,z are values in memory addresses
             cmpw cr0, x, y   // store the condition in a condition register
             bne cr0, loc_000000C  // branch if the condition is not met
             addi z, z, 1     // increment z (when the condition was met)
loc_000000C: --------------------
                              // Without Branches:
             subf r0, x, y    // r0 = y - x
             subf r3, y, x    // r3 = x - y
             or r3, r3, r0    // r3 = r3 | R0
             extrwi r3, r3, 1, 0   // extract the sign bit (here equivalent x != y)
             xori r3, r3, 1   // flip the bit
             add z, z, r3     // increase z by the flipped bit
```

Figure 2.3: Two different ways of implementing the C statement: `if (x == y) ++z;`

2.2.2 Hardware

Virtually all processor architectures include registers with extremely short access times, in which intermediate values of computations (both on addresses and actual data, sometimes separately) are stored. Also, most systems use a first-in-last-out structure, simply called the stack, to manage the trace of functions calling each other and passing on parameters, while sometimes registers are also used to store parameters. There can be special purpose registers that point to the current top of the stack, the next instruction to fetch and execute, the next instruction that is to be executed upon returning from the current function, the current data location that is iterated on and so on; furthermore some registers can be reserved for only a certain type of data, certain intermediate values used in arithmetic operations, or flags that carry information from one instruction to the next and make dependent jumps and therefore conditional blocks and loops possible. Some designs recommend using one specific register as the accumulator which stores the current computational result to be used again in the next operation, much like the value displayed on a simple calculator, or even force such usage by providing instructions that implicitly use a single register as operand and output storage.

2.2.3 PowerPC

The systems to be analyzed in this thesis were built for the PowerPC family of processors [8]. Notably, PowerPC was conceived as a RISC (Reduced Instruction Set Computing) architecture, which implements separate opcodes for accessing memory and arithmetics (as described by the term “load/store architecture”) and few composite instructions that take many CPU clock cycles to execute and are variable in length. PowerPC instructions are always 32 bits in length and consequently, as an example, loading an immediate 32-bit value to some register takes two lines of assembly code, see Figure 2.4. Also, jumps or other references to distant locations in memory also require multiple instructions, while a single variable-length instruction with versatile addressing modes would be able perform

such operations by itself.

```
li r3, 0x5678         // load immediate
addis r3, 0x1234      // add immediate shifted
```

Figure 2.4: Loading a 32-bit immediate value within fixed 32-bit instructions

PowerPC calling conventions usually mandate that arguments are passed via a certain section of registers, although if the register space does not suffice or some parameter registers are in turn used when calling another function by the callee, or in order to facilitate debugging, a special section on the stack frame of the function setting the parameters is used to extend or back up the parameter registers. This means that functions often contain numerous instructions at the beginning and end, who are collectively called prolog and epilog, respectively: The prolog saves the return address on the stack, backs up volatile registers (those that may be overwritten by the called routine) if necessary and allocates stack space, and then retrieves arguments from registers and/or the stack; the epilog restores the former stack pointer and instruction pointer, after the function typically has placed its return value in a specific register. A typical function's prolog and epilog is shown in Figure 2.5.

```
mflr r0               // move the return address from link register to r0
stw r0, 0x0008(r1)    // store it in caller's stack frame
stwu r1, -0x0038(r1)  // create a stack frame

[function body]

lwz r0, 0x0040(r1)    // copy the saved return address to r0
mtlr r0               // move it to the link register
addi r1, r1, 0x0038   // remove the stack frame
blr                   // branch to link register (return)
```

Figure 2.5: Example prolog and epilog, where r1 holds the stack pointer

Both of these phenomena that occur in PowerPC code exemplify the small scale idioms that make up a large part of machine and assembly code, especially in RISC designs. These will be later discussed in Section 3.2.

2.2.4 Case Study

The particular system that serves as motivation for this thesis is an embedded piece of software implemented on a PowerPC architecture. It is likely not performing extensive arithmetic computations, which could result in extensive duplication between the delegating, managing and communication activities of the program and little actual redundancy in functionality or easily recognizable, unique pieces of algorithmic code. From its memory segmentation layout it is evident that space was not a limiting factor in the design of

the software, so the code has likely not been optimized specifically to fit a restricted size in memory. It is known that there is a bootloader within the system, which naturally shares a significant portion of code with the deployed executable program, and as will be explained in Section 2.3.1, matching the functions from one part with their respective copies or equivalents in the other is closely related to version control and therefore, this study subject offers an interconnection between reverse engineering and that field of software engineering.

2.3 Clone Detection

One of the metrics that can be applied to software is how much duplicated, possibly redundant information it contains. Often, a set of similar fragments can be generalized within the software system and thereby be expressed more concisely and also taken out of specific contexts to form a distinct unit, usually a procedure or class. This is a way of structuring or "re-factoring" [9] the system and is a primary motivation for using higher-level programming languages, as it works towards a program representation closer to the conceptual grasp of the programmer. Also, the separation of concerns in the code, as in this case removing a fragment from several different surroundings to exist as a unit independent from those places in which it currently is used, makes it easier to find these places by the unit's name, and easier to manipulate the operation of all of them by editing the singular unit that has been "extracted". Sometimes changing the structure of the system is not possible or practical, not easily automated, or detrimental to its efficient performance (or simply not the goal of development, as in reverse engineering, see Section 2.3.1). In these cases, still, knowing about redundancies in a system improves a developer's understanding just as the actual change reflected in the code base would [10, 11]. Conversely, undetected and undocumented duplication can lead to internal ambiguities or inconsistency in the behavior of the system as soon as one instance is changed independently of the others. Overall, the quality of code immediately benefits from recognizing and re-factoring duplications in many respects, including understandability and thereby maintainability, re-usability and testability.

As the detection of duplicated patterns (commonly called "clones") can be followed by improvements to the code base — carried out either by a human engineer or intelligent algorithms — and consequently is a research field of growing interest in software engineering as growing processing capacity accommodates extensive analyses on large software systems. Numerous tools and methods have been proposed for finding clones, especially for imperative high-level and intermediate languages, but also both highly abstracted models and extremely low-level machine code. Depending on the instruction or description language analyzed, more or less universal measures of similarity can be employed, of which some possibilities are explained below in Section 2.3.3.

Comprehensive reviews of the state of the art are abundant, many of them trying to measure the effectiveness of the proposed tools in different contexts [12–17] (a brief summary is given in Section 2.3.3), for a discussion of a range of potential applications see [11], as well as Section 2.3.1. Recently, many researchers have expressed the need for

a standardized benchmark for evaluating these tools, as opposed to evaluations based on subjective definitions and truths to compare with [18–20]. The algorithms presented in literature have been applied in dozens of commercially distributed analysis tool sets, but while theory has already been successfully expanded to diverse types of subject systems, few immediately ready-to-use tools are available at all, academically or commercially, for use cases beyond either popular imperative languages or plain text comparison (which does not use information about the type of input it receives), see [21].

2.3.1 Clone Detection in Reverse Engineering

The motivations for clone detection research are not limited to condensing information to fit space, or refactoring and separating concerns in a system in order to make it more maintainable. These are all improvements to a system that can be made during its construction, however, as mentioned above, clone detection can provide understanding of the functionality of software as well, which is the goal that for example reverse engineers work to achieve. Clone detection on its own can not reverse the process of compilation to obtain source code that could have been used to generate the target code under examination. In combination with certain kinds of external truths, though, the same abstraction techniques prove useful for identification and reverse engineering of software of possibly foreign origin. For example, detecting malware [22], as well as libraries included in a software system (which often involve license violations [23] and can be redundant), and comparing versions of a system [11] is made possible by algorithms that match similar or even copied code fragments.

Malware The techniques employed to let malicious code pass protection mechanisms undetected become more and more sophisticated. While viruses and similar schemes rely on excessive duplication across a network to achieve a sizable goal, they also become immune against signature-based and other superficial ways of recognition by continually changing themselves. When comparing candidates with known specimen of malware, parts of their code need to be examined separately, since the mechanisms by which a particular kind of malware propagates, changes and profits are unlikely to be completely overhauled (but they still can be elaborately obfuscated). Here the available techniques of both malware recognition on a coarse level and reverse engineering on a detailed level can be augmented by clone detection.

Libraries Software re-use can lead to situations in which a system incorporates a number of compiled collections of code whose subcomponents might never be examined by the developer. Without reverse engineering the machine code, the developer can not know what other functions the system can perform by virtue of the included libraries that are not directly or indirectly presented in the interface exposed to him, and thus, he may re-implement those features or include the same or similar libraries another time in his project. Detecting the cloned libraries is usually easy, especially when the exact same version is used in both cases, and avoids this unnecessary effort.

Licenses Additionally, the author of some component may restrict its use in building a software system to applications or business models of a certain kind, or require proper attribution or re-licensing. As soon as a programmer incorporating a library from its original source willingly misrepresents the license in his own software package, the user of that package in turn would have to analyze its internals to know what terms he has to comply to. The original creators of licensed software are interested in finding intentional and accidental misuse of their work, and since libraries are mostly included unmodified — as they are meant to, searching software repositories for license violations has become practical in some cases.

Versions Another domain in which highly similar fragments of code are expected and looked for is the comparison of different versions of the same system. The identification of fragments that correspond to each other between the states of a system at different points in time is crucial in collaborative editing and version control systems, and for documenting the effects of the changes made for the purpose of compatibility with other systems. Depending on the intricacy of the editing that has taken place, this can be a considerably complex process. Clone detection, then, is the part of this process that consists of finding parts that have probably not changed or only changed superficially (are unlikely to have been deleted at one place and independently re-created at another in such a similar form as to be detected as a clone). While version control is the most evidently and efficiently applied in the context of managing source code repositories (and relatively primitive clone detection, along with closely related techniques, is widely used for that purpose), comparing versions of binary libraries or executables may also be necessary to spot relevant changes if source code is not available. Strictly speaking, the latter application belongs to the field of reverse engineering, as information about such changes is intrinsically on a higher level than the analyzed code, and from it conceptual changes in the implemented functionality can be inferred.

The use case for which the suitability of clone detection techniques will be evaluated in this thesis is consisting of both general comprehension and comparing versions of a system. With the use of bootloaders and customized software versions, it is known beforehand that large components will be almost perfectly duplicated, and in that sense, comparing different versions, states or representations of an embedded system serves the purpose of documenting its structure and behavior. The problems that the nature of machine code languages imply, especially for identifying matching parts of different versions of a system (where usually a low quantity and spread of changes is expected), are discussed in Section 3.2.

2.3.2 Clone Types

Code clones can be differentiated, among other criteria, by their similarity, that is which kind of operation is necessary to convert one instance of the clone into the other, even if duplication and modification is not the way the clone originated. One of the most prevalent categories are clone types 1 to 3 as defined by Bellon et alii [13], and by common extension 4 (several reviews cite papers using this term [15, 16]), with other

terminology often mapping well onto these types [15]. An overview is given in Table 2.1, distinctive examples are listed in Figure 2.6 (inspired by [16]) and the significance of this particular system of classification is assessed in the following.

Other criteria include the relative position of the clone instances to each other and their size. For instance, cloned function bodies, beginnings of functions are distinguished, as well as small blocks, loop, switch and conditional clones, or clones contained within one module, file or even function versus those scattered across separate parts of the system [24, 25].

I	layout, comments
II	identifiers, literal data, types
III	lines added/removed/reordered
IV	same semantic

Table 2.1: Clone types I to IV

Assessment It should be noted that only the first two clone types are rigorously defined, while distinction of type III clones from type IV clones or non-clone similarities is hard to make without a more or less arbitrarily chosen quantitative threshold on a distance measure [13]. There are modifications within individual lines of a piece of code, depending on the programming language, which therefore would be recognized as a removal and addition of two lines by a type-III-capable detector, that do not change its way of operation. For example, in C, incrementing a variable is possible by at least three equivalent statements that contain a different number of tokens (here: identifiers and data) and would not be considered a type-II-clone, see in Figure 2.7. This principle also applies to assembly code, it is by no means a unique representation of a computational process. As Figure 2.8 (based on Figure 2.4) demonstrates, there are different ways in PowerPC to set the most significant 16 bits of a register when the least significant 16 bits have already been set by an instruction like `li`, which has set the former to zero: One can XOR each bit with the 0 that is already in place, or add the value to the zeros with no carry occurring at all.

Then, if potential normalization steps applied to the code do not equalize these modifications, the threshold for similarity of lines may not be reached and the clone is not reported at all. Detecting semantic clones of any kind requires extremely sophisticated "normalization" even across lines, in fact on the scope of the whole clone instance, to ensure that any change that is not affecting the functionality is evened out. This is equivalent to reasoning about semantics and thereby specific to each environment and language, and in fact only shifts the problem into the space of models representing that functionality, in which then another measure of similarity has to be devised.

It is also impossible to generally establish a qualitative criterion based on the type of a clone that determines whether editing turned a copy of one instance to the other instance's location into what has been found similar, or if they were written independently from each other. This complicates finding the prevailing causes for cloning in a given

```
void foo(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
        bar(sum);
    }
}

void fooI(int n)
{
    int sum = 0; //zero element of addition
    for (int i = 1; i <= n; i++)
    {
        sum += i;
        bar(sum);
    }
}

void fooII(long count) {
    long s = 0;
    for (long i = 0; i <= count; i++) {
        s += i;
        bar(s);
    }
}

void fooIII(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        printf("\%d",sum);
        bar(sum);
        sum += i;
    }
}

void fooIV(int n) {
    int sum = 0;
    int i = 1;
    while (i <= n) {
        sum = (i*(i+1))/2;
        bar(sum);
        i++;
    }
}
```

Figure 2.6: A code fragment and type I to IV clones of it

```
{
    x = x + 1;          // three tokens
    x += 1;             // two tokens
    x++;                // one token
}
```

Figure 2.7: Three alternative ways of incrementing a variable in C

```
li r3, 0x5678          // load immediate
addis r3, 0x1234       // add immediate shifted

li r3, 0x5678          // load immediate
xoris r3, 0x1234       // XOR immediate shifted
```

Figure 2.8: Two alternative ways of loading a 32-bit immediate value within fixed 32-bit instructions

system, since some independently created clones are mistakenly identified as type I-III, as they coincidentally remain similar even under the syntactical constraints imposed by the first types of clones, and could be assumed to have been copied. Conversely, clones that actually have been copied and then sufficiently edited may not exhibit the arbitrary degree of similarity required to be considered more than a semantic clone, and then are hardly distinguishable from other semantic clones that were introduced by recreating existing functionality in a distinct way. The implicit assumption that is sometimes made [26], being that there is a correct partition of clones into more and less syntactically equivalent ones that reflects the way they originated, is not universally true.

Clones within functions are rarely explicitly dealt with in literature so far, possibly because they are often insignificantly small and offer little insight for the purposes of either re-factoring or reverse engineering that could not be gained effectively by examining functions in isolation. Distinctions based on the file structure of the system can be of practical relevance when incorporating the changes made following clone detection, or when measuring on what scope clones are usually introduced in the software production process. However, basing the actual clone detection process, aimed at the improvement of code, on the original development decisions whose result is analyzed hinders that very process. Clones between similar positions in separate functions are found using fine-grained graph-based analysis (see Section 2.3.3).

2.3.3 Detection Algorithms

In the literature, several approaches to clone detection are differentiated. They are largely characterized by what kind of transformations they apply to the code to be analyzed and what form of representation they then proceed to find duplications in. Some natural demands on the programming language of the subject code as well as the granularity of the search result from what representation is used. In addition, the approach of a clone

detection tool often implies what type of clone it can find, as detailed in the following.

Text There are text-based detectors that generate hashes for segments of the raw code, usually lines, and sort them into buckets and report any bucket filled by multiple lines [27]. After the development of this simple approach, Ducasse et alii have proposed printing the incidence matrix of these matching pairs as a "dot plot" for visual analysis [28], which would be able to identify even type 3 clones with some lines removed, added or switched as diagonal lines parallel to the main diagonal of the resulting square image. (Even this step can be automated by dynamic pattern matching as mentioned in [16].)

General-purpose tools such as "diff" (as introduced in e.g. [29], the underlying algorithm has been published in [30]) and "Simian" [31] operate similarly, as they require no specific syntax except regular line breaks that result in meaningful matched lines after comparison, which is the most obvious convention in programming or in fact text processing overall. "diff" and its variants and extensions are famous for their ability to visualize the shortest and therefore reasonably likely set of editing operations that changed one version of a file into another, by looking for longest common subsequences (LCS) among the matched lines or other items. It therefore takes into account the order and sequence of lines and allows for slight changes, on the line level, in code and readily presents contiguous sequences of matches, but can also exhibit a high worst-case algorithmic complexity and is not always practical for large files with complex changes having been made (especially, of course, if these changes concern the content of many individual lines, such as replacements). Derivative algorithms are widely used in version control systems; its significance for the application scenario of clone detection is explored in [32]. While "Simian" automatically or optionally excludes certain parts of the program, such as module/package management commands, comments, delimiter characters or literals, it is not a full-fledged lexical analysis tool, because it still performs a comparison on the raw string lines of the source code.

Tokens Token-based algorithms like "Dup" abstract away identifier names and the concrete values of literals [33], while syntax-based algorithms analyze the syntactical composition of the code [34]. Both of these methods employ a lexical analyzer ("lexer"), refining the granularity of their analysis to the token level as opposed to lines, although lines (e.g. in "Dup") and other language-specific delimiters (see below) are sometimes respected in the detection process to make sure the matched token sequences represent some elementary syntactical segment, or at least when reporting these matches to the user. Here, refining means it is now possible to relate lines to each other based on their similar token sequences, while comparison on the scale of whole lines would consider them as different as any two lines can be. While there are algorithms that compare text strings on the level of individual characters, similarity between tokens in their characters are usually of no interest to finding program code with actually similar functionality, as this is exactly the way tokens in a programming language are defined: unitary elements of meaning in a language represented by a certain class of strings, or lexemes, in code (exceptions include `elif`/`elseif` in e.g. Python, similar to `else` and `if`, whose meanings

combined it expresses exactly). In addition, the resulting token sequence could be filtered to exclude or equate less meaningful kinds of tokens such as access modifiers like "public" and "private", or types that are very similar to each other [35], and normalized to enable matching sections of code that use different sets of identifiers and values which however can be mapped to each other injectively, as for example "Dup" does. Token matching sometimes tolerates insertions and deletions inherently by performing LCS search, an example is the sophisticated plagiarism scanner "JPlag" [36].

Trees Syntax clone detection then additionally uses a parser to structure the sequence obtained by the lexer in tree form, just as a compiler would do to understand the structure the author expressed within the syntax of the programming language. This prevents the algorithm from finding clones that do not correspond to syntactical units, and as such are likely to be incidental and irrelevant or at least hard to re-factor (save by usage of macros, which defeat the purpose of making the code more understandable altogether). Alternatively, token-based detectors have to be accompanied by pre- or post-processing which requires knowledge about the delimiters used in the programming language and therefore more than its lexical rules, to avoid such useless clones [16]. Recently, suffix trees or suffix arrays are often used to match tokens sequences [15] and even syntax trees [37].

Graphs The next step further in abstraction is the construction and analysis of the program's dependency graph, which may not only contain statements and expressions as its nodes, but also variables and procedures. They are connected based on various relations which they bear to each other, for example the value of a variable being computed in a certain place in the code, and next used in another [38]. Another structural feature beyond the syntax of the analyzed code, the flow graph within a function, can be used to find cloned functions. The analysis of these graphs allows detection of very high-level clones which either rely on the same external resources and operate in the same context or share some general manner of operation, ostensibly then also the goal of their operation, but whose details are written down in a different manner. Thus, it uses semantic information about the code and is able to find non-contiguous and even simple semantic clones that occurred by independently creating multiple representations of the same algorithm or use case, but is less scalable than analyses on simple syntax constructs [15]. There is a commercial tool ("BinDiff") that combines the comparison of the flow-graph between the basic blocks of a function, entry and exit points, and relative positions with the comparison of the instructions themselves [39]. It is therefore able to evaluate clones based on the position (both in the simple numerical sense and in terms of possible execution order) within a function they occur at.

Metrics The most large-scale techniques use some of the mentioned representations of code, such as the original raw text separated in lines, the token sequence or the syntax tree the tokens of a function form, and apply various quantifications to segments or subtrees of them to obtain a vector of values that characterizes these elements [40, 41].

(The conversion functions used to obtain the values of these vectors are commonly called "metrics", this usage is not to be confused with denoting the metrics used in software engineering at large, of which the degree of duplication or "cloning" is but one.) Comparing these vectors yields information on which of these elements could have been copied and pasted, or otherwise constructed similar, and is also invariant to ordering of statements, since the metrics employed usually just concern counting occurrences of calls, loops, inputs and other features of individual statements. However, this approach is oblivious to duplicated fragments any smaller than the elements examined [15]. Consequently, these metric-based algorithms are often employed on a file level to find copied software components, or copies of malicious software, of which a specimen is known and can be subjected to metrics (see Section 2.3.1).

Miscellaneous Several other approaches have been proposed, including a hybrid one of applying metrics to syntax trees, and some other algorithms relying on neural networks or data mining techniques [42, 43].

Language Some researchers consider specifically analyzing what others factor out, namely natural language elements such as comments and identifiers, which hint to the semantic "meaning" of the code they occur in [44,45]. The researchers report success in using latent semantic indexing, which is a technique of information retrieval on a corpus of natural language that infers the meaning of words and phrases from the context in which they are used. They also note that it disregards the order of appearance and the composition of words (such as a variable "hitCounter" most likely being related to another named "hitCount" at a different location, but less so to one named "missCounter" or even less to some "loopCounter", despite being used in almost the same way), which could enable even more sophisticated analysis of names and phrases. This kind of algorithm is especially powerful in detecting copied code fragments, because most of the time comments and some identifiers are not modified at all, while only contextually relevant identifiers such as input and output have to be changed, and may beneficially enhance existing structural approaches like the prevalent well-established token, syntax or dependency analyses. The aforementioned tool "BinDiff" also uses the names of functions to spot potential duplicates, in combination with several other criteria.

3 Reasoning

3.1 Granularity of Reverse Engineering

In the application scenarios mentioned above (see Section 2.3.1), detection algorithms need to be developed or adapted to work on binary code instead of "human-readable" programs, the former lacking many of the modular, grammatical and scoping structures of typical source code languages. In terms of layers of abstraction, binary instructions are further away from the concepts a developer might consider and look to find (duplicated or not) within a piece of software. Several mechanisms exist in compilers and interpreters that "instruct" a machine exactly how to carry out the algorithms and tasks defined in a higher-level language, and the concrete implementation of a program on the scale of individual processor cycles is usually hidden from the developer. Therefore, clone detection has to either work on a scale large enough so that these details do not interfere significantly with the results it provides (as for example whole libraries), or will likely be only able to be used productively by engineers with experience in processor technologies and their assembly languages. On such a low level, the peculiarities of such languages have to be considered and incorporated in the clone detection process, so the algorithm finding clones, just as its user, has to work with a lot less information density and more idiomatic constructions, flatter structures and no explicit encapsulation.

Generally, the more restricted the search for clones is by introduction of a truth by the problem domain, such as related versions, malware samples or libraries, the less fine-grained does the algorithm have to operate, and consequently the less does it have to take into account the way the subject system's language is structured. The most restricted example would be simply verifying the exact equality of two files or even directory trees, which can be done simply by going through them bit by bit in order, or employing a hash function to condense the contained information of the reference file to quickly check the identity of any other with a certain confidence. This approach is universal, as it requires no prior knowledge about the contents of the files at all. The opposite scenario, with no meta-information or reference for comparison, is pure reverse engineering (which can be seen as the equivalent to program comprehension on machine code level) and requires a thorough analysis of individual units of code. The possibility that assembly code is identical by copy-and-paste still exists as long as the original source code contains such duplications, but the hash-based algorithms designed to find them are too primitive to assist a reverse engineer or programmer trying to understand how a given piece of assembly code works.

3.2 Assembly Language Structure

Machine language and any representation of it is linear apart from partition in functions. That is, the sequence of instructions may implicitly contain a conditional block or loop via jumps, or create scopes for variables by using the stack, but does not explicitly

specify a hierarchy of levels at which the instructions are located (see Section 2.2.1). Therefore, conversion into a flat syntax tree is useless, and the only tokens that could be considered are operator and operands. The conversion in a token string would, however, not incorporate the restriction on how lines are formed, i.e. by exactly one opcode and a limited number of operands, and simple text-based approaches do not even use the separation of elements (actually opcodes and operands) at all. It is obviously not harmful and most likely useful, if only improving performance or eliminating a certain class of false positive detections, to use the knowledge all of the structural constraints of the subject matter. The likely best technique would be one that is specifically geared towards the way binary code is structured.

The only organizational units that are semantically meaningful and therefore lend themselves for the system to be partitioned in are either functions or instructions. Functions are the most useful for understanding the workings of code, both because, presumably, the underlying source code is organized modularly in sensibly named functions by its author which largely correspond to those appearing in the compiled code. Furthermore, the stack, parameters and variables and the current "meaning" of a register are limited to the scope of a function and thus the functionality of a function is self-contained enough to be relatively easily described and reasoned about as a concept in isolation. But functions in machine code are not created by explicitly copying others and thus do not need to exhibit any degree of duplication on the whole beyond that of the corresponding source code units on the whole, which may have been copied by programmers. Since the reverse engineer is interested in their purpose, a look inside those functions is necessary to detect differences that do not impair the commonalities between them that represent commonalities in the functionality they implement.

3.2.1 Instructions

Single instructions, on the extreme end, are far too small units to be sensibly compared to each other, even with reductions and filters in place. When one compares them, including their arguments, one can reconstruct possible relations between functions by the number and nature of shared references to external data or other executable code blocks, and what emerges is a needlessly indirect way of classifying functions by their dependencies as a purely dependency-graph-based analysis would do (see Section 2.3.3). This is because data or code is referenced by only a few instructions in all of the program, so the mode of operation used on them does not add much information on top of the fact that they are referenced. But such a system of classification can already be too specific for many clones. Duplicate functions that differ from each other mainly in the particular data used, or the parts of the program they pass on control to as a consequence of being situated in different execution paths, are not found by such an approach.

After abstracting away these references in each instruction individually, however, only operators with their operand types, which are hardly unique pieces of code, remain. As most operands represent a low-level concept such as adding one value to another (see the examples in Section 2.2), they usually can not identify any specific kind of function or context by themselves. Counting the occurrence of features such as operators

or operands in a function can provide useful indicators of its purpose, as for example many multiplications and additions of non-constant values probably appear in a function that carries out some arithmetic computation, or repeatedly incrementing a register combined with call instructions indicate that a branch table is traversed. But in general, functionality results from a sequence of instructions and the operands they share among and pass on to each other, and this sequence may or may not encompass the whole function.

3.2.2 Instruction Sequences

Several algorithms have been proposed to find duplicated substrings or subsequences among lines of text, with more or less tolerance for inserted lines not belonging to the clone, a prominent example being "diff" (see Section 2.3.3). The tool "ACD" [46] is actually geared towards interpreting machine instructions as lines [47] is this respect. After normalizing the items in question, those are treated as atomic tokens, and any changes to a certain instruction or token that remain are not tolerated any more than a completely different line in its place is tolerated, that is as an interruption of the matching sequence. Unfortunately, as detailed in Section 2.2, such differences are frequent in assembly code: Often the same idiomatic constructions are built using different callbacks, input values or sources, and sometimes under different circumstances a compiler (even if it is the same compiler at all between multiple versions or even components of the subject system) arranges a piece of code in a slightly different way to optimize pipelining performance, reduce data access times, align code or save a few instruction cycles. It would be beneficial to be able to assess the degree to which instructions are similar, without inspecting each in isolation and then comparing every possible pairing among them.

Also, the problem of aggregating information about instructions to information about fragments of code becomes complex starting from just a pair of instructions, because the discrepancies found so far have to be continuously kept track off for a large number of instructions ahead and in front of each pair until the algorithm reaches a threshold by which any reordered or otherwise modified clone including the initially found pair is sufficiently unlikely. Iterative algorithms that begin analyzing units as small (in terms of information density) as single assembly instructions have to optimize the representation in which the matching pairs are presented to the user to include the most probable of the matches in the reported matching fragments. The sheer number of matched lines soon becomes unmanageable in larger collections of assembly code, even when only exact matches are reported, because in such a limited instruction set with several irreplaceable instructions with few (sensible) arguments as machine languages are, some lines are unavoidably copied all over the code. This is especially, but not exclusively, true for the beginnings and ends of functions, in which often the same tasks concerning the stack, parameters or the temporary depositing of register values in memory have to be performed.

3.2.3 "Sliding Windows"

The limitations of assembly code structure leave the engineer with either metric-based techniques similar to the ones mentioned in Section 2.3.3, or order-sensitive matching algorithms, in case they already provide enough coverage and information for further analysis of the software. When applying metrics to their subsequences instead of functions on the whole, the precise sequence and order of instructions is not rendered irrelevant on a larger scale, but disregarded within the length of those subsequences, or "regions". An algorithm comparing (ordered) source code regions directly has already been proposed early on [27], along with strategies to reduce the sheer number of regions to compare, respecting line delimiters and maintaining (overlapping) code coverage while still allowing for regions of varying length.

However, to not miss any clone (of some minimum length) that would not be reported when only captured by a reduced set of regions in overlap with other parts of the code, every possible region inside a function of that length has to be included in the analysis, as suggested in [48]. Here, the subsequences of the sequence of instructions in an assembly function are analyzed, and as such can be of any uniform length without compromising the usefulness of information about them (unlike in the case of character strings purposefully split into lines which the above source code window approach deals with). Retaining the same region size across the system makes the comparison steps which follow later easier to carry out, and the term "sliding windows" is used to describe the collection of equal-length fragments of code from the beginning to the end of a function. Also, the paper tests a unique way of normalizing operands to obtain only information about "what the code does", as opposed to what values it operates on; this approach matches the reasoning in Section 3.2.1. The implementation of Sæbjørnsen et alii is not publicly available, but will be reconstructed and modified as detailed in Chapter 4, and evaluated in terms of precision, recall and performance in Chapter 5.

It is to be determined what type of clones can be detected by region- and metric-based algorithms, and it is not trivial to decide how far operands should be abstracted, because some like the stack pointer or source index carry semantic information and are not interchangeable. Information about the special meanings of certain registers or instructions is dependent on the process architecture the analyzed code is written for, and the quality of detection incorporating this knowledge may vary considerably. Also, the semantic meaning of instructions could be incorporated, as for example jumps are treated specially as the delimiters of basic blocks (see Section 2.2.1) when analyzing control flow inside functions (this alternative or supplementary approach is discussed in Section 3.3). Certain instructions that appear excessively often in the code in the same places such as function prologs (see Section 2.2.3) or before calls to other locations and add little information about the utility of a function could be simply ignored by the detector or not considered similar unless they use the same operands or even appear in the exact same sequence — which would suggest that they actually are used in the same context. Unfortunately, these analyses again require good judgments based on knowledge about the platform architecture, and also add elements to the detection approach that were dropped because of possible problems as explained in the previous Sections.

First, it would be useful to conduct experiments on the nature and quality of results of comparing assembly code regions in a general, normalized form. Since compilers often optimize quite freely, not to mention intentional obfuscation by the authors, the order of instructions is often tweaked, on the other hand they produce such idiomatic code, whose passages are hardly unique to the particular function the code performs, that disregarding structure altogether and simply measuring occurrences of certain features could prove too unspecific.

3.3 Alternatives

Similarity measures relying on either the control flow graph internal to each function, or the inter-functional dependency graph of a system, promise to give a high-level estimate of the complexity of each one's functionality and its place inside the program. However, simple functions require examination below the level of basic blocks, and in large systems even moderately complex graph layouts are likely to be duplicated by chance, without any semantic connection. Also, due to the nature of assembly code, in many cases control flow and therefore the overall organization may not be obvious and definite as soon as data fields are referenced for determining jump targets. This is often used to select functions to call during execution time, so the set of possible addresses in the data field at the time of access is dependent on circumstances and computations not local to the function reading the data field, and potentially huge and thus a lot of the flow paths taken into account in the analysis. While the combination of both these and general instruction-based approaches is powerful, matching clones based on non-control instructions, which are more directly influencing the function's return values and side effects, alone could already provide the utility reverse engineers need. Potential shortcomings of algorithms which ignore flow structure and dependency relations, as the chosen one does, are identified and discussed in Chapter 5.

Binaries, or even disassembled binaries, also do not offer comments or identifiers by themselves that explain the code. It can not be assumed that a symbol table comes with the raw code, or that the disassembler recognizes any common patterns generated by the compiler and assigns meaningful names to them, so the only semantic information already present are strings. Grouping functions by what strings they use (these strings may even be human-readable or known to the user from an external context) is a useful approach for analyzing binary files such as executables or libraries, but completely depends on the usage of actual strings in the code and as such is not universally applicable.

3.4 Clone Types

The usual way clone types are defined (see Section 2.3.2) is not immediately applicable to assembly code, but parallels can be pointed out and categories of assembly clones could be established. Firstly, there are no comments or other non-functional elements, and any layout of a piece of assembly code has been generated by the disassembler to make it more readable, so type I clones correspond to exactly identical fragments of code.

Secondly, the distinction between type II and type III clones is problematic, as the notion of types does not exist in assembly in the same way as in higher-level languages. There are addressing modes, which are signified for each operand to "tell" the processor how to interpret the raw number that represents it: literally as the value it is, as a reference to a location in memory, or a special code for a register or the stack, possibly from a certain offset. The inclusion of clones with a changed order or sequence of instructions is immediately achieved by subsequence (as opposed to coherent substrings) search algorithms and those operating on metrics only, instead of the instructions at their precise position in a function. The usefulness (and possible harmfulness) of this more tolerant approach is an important consideration. The quality of findings presumably depends on the setting assembly clones are sought in: If significant changes in the instruction sequence occur frequently in the places of interest (which are not known beforehand) or if the desired information can already be derived from the results of a more strict algorithm. The algorithm described in Section 4 is in theory arbitrarily permissive of changes in both the opcode sequence and the operands, but can be tightened to not allow different instructions to appear in the matched windows. The order inside a single window is always ignored, but when overlapping sequences of windows match, there is also a high chance (in the tightened version certainty) that the matching instructions also appear at the same position within the detected clone region and relative to each other.

Thirdly, the fourth category of code clones is generally still hard to define precisely. Finding semantic clones might be possible even in these low-level languages, especially when strings and external functions and data are referenced. Also, as explained above (Section 2.3.2), symbolic execution or transformation in an extremely abstract form can provide high-level information as the logical continuation of "normalization" efforts on the code, just as in clone detection for source code. This is an active field of research, but prohibitively complex and architecture-specific in relation to the scope and purpose of this thesis. It is not easy to make definite statements about the supposed "meaning" of a function or other piece of software, and this is in fact the process of analysis that follows the application of relatively low-level clone detection tools.

4 Algorithm

The algorithm which was implemented (in Python 2.7) and is evaluated in this thesis is closely related to the one presented by Sæbjørnsen et alii in the original paper [48]. Notably, it uses a different kind of locality-sensitive hashing (see Section 4.3), a more rigorous post-processing step reducing the result set, and optionally a modified normalization step. The required input is a disassembled executable, containing clearly delimited functions and decoded instructions, and the output is a list of similar instruction sequences as denoted by the function they were extracted from and their position therein. This locality information is also aggregated to concisely list the candidate pairs of functions with the highest ratio of "cloning". As with the above authors, IDAPro is used to produce a well-formed disassembly here, but it is not required to specifically use IDAPro in favor of any other advanced disassembler capable of decoding the given instruction set.

The whole process can be roughly divided into four steps, the end result of which are, in order: normalized code regions or windows, buckets of those windows sorted by similarity, clones filtered from these buckets, and function pairs derived from the cloned regions they contain. This setup largely corresponds to the inexact clone detection algorithm described in the original paper. Additionally, an interface, using IDAPython, extracts the disassembly created inside IDAPro in a well-structured form and also writes back information gained from the clone detector into the disassembly file. Each step, as represented in this thesis by a Section or Subsection, is carried out by a (with few exceptions) independent script file and the data flowing between them (including said input structure) is stored as JSON [49] in text files in the same directory as the disassembly took place, Thus, all of the steps can be performed in isolation and parameters can be varied for each one individually. (In the following sections, I will use common programming terminology such as that of Python, in which I implemented the algorithm, instead of JSON terms even when describing JSON structures. For example, dictionaries and lists would in JSON be called "objects" and "arrays", respectively.) The file names share a common stem, to which the specific type of intermediary or result data set is appended, as well as some parameters that have been set in any of the previous steps.

4.1 Input

Parameters string output file stem by default, the name retained in the IDAPro file

The script providing the input to the clone detection runs inside IDAPro and has access to its internal data structures. It first looks for all "heads", which are any defined elements inside either code or data regions, and every one that is marked as being code is recorded in a JSON data structure as well as dumped in a string representation of the assembly program. Any new function that is encountered (each head can point to its containing function, if there is any) is recorded in a separate dictionary, along with

its absolute starting address and length in bytes. To be precise, the "mnemonic" of each instruction, which is the textual symbol for the operator, and by extension, specific operands also included in the opcode, in assembly code, constitutes the first element of a list which represents the instruction, along with a list of operands designated by their type, actual value in binary and the disassembly interpretation stemming from these. Now, the JSON data structure, which will be further processed, is a dictionary which maps function names to their respective ordered lists of instructions (each of which is structured as described above). This approach disregards instructions not belonging to functions, as they are likely not completely disassembled and delineated, and treating them as identified code and grouping them sequentially in functions would be speculative and possibly ambiguous. This JSON dictionary is "dumped" in a text file specified by the stem given as a parameter.

4.1.1 Joining

Parameters	string	input file stem 1	
	string	input file stem 2	
	string	output file stem	by default, the input stems combined

If two versions of one software are to be compared, or the subject system is spread across multiple files (which then result in multiple IDAPro disassembly databases, which are not trivial to merge), the algorithm has to first "export" the data from separate instances of IDAPro and then join it to continue the analysis. The string `_old` is appended to the keys, i.e. function names, in both the index of functions and the code database of one of the disassembly files. This feature resolves name collisions and later allows for manual or automatic detection of direct clones between corresponding functions from different versions of a system (see Section 4.4), or even more complex patterns of clones between a function and others across versions. More than two input files can only be processed by nesting these joining operations, so that function names end with multiple `_old` strings and only function pairs from "neighboring" files (in the order of joining, which differ in one `_old`) are compared, but this mechanism can easily be replaced by a more scalable one in the future.

4.2 Windows

Parameters	string	file stem	
	integer	window size	any integer > 1
	integer	window stride	1, not modified in this thesis
	integer	maximum operands	4, covers all cases for PowerPC
	boolean	combination features	True by default, False by modification
	boolean	anonymous normalization	as above, vice versa

The first operation is the partition of functions using the "sliding windows" approach (see Section 3.2.3). This partition depends upon the desired size of the windows (note that functions smaller than this size will not be represented at all), and the step size or

"stride" between the windows. Generally, this results in $((function length - window\ size + 1)/stride)$ windows, rounded down, and adjacent windows overlap in $(window size - stride)$ instructions. For example, a function of 12 instructions can be partitioned with a stride of two into five 4-instruction-windows: 1–4, 3–6, 5–8, 7–10, 9–12. However, as explained in Section 3.2.3, a stride of more than a single instruction incurs the risk of missing a clone, even though it would have fit into a window of the chosen length and be reported as one. The reliability of a version of this algorithm that periodically skips windows but still covers the complete code with overlap could be assessed in the future.

The normalization step applied to the instructions in the window to obtain a feature vector for inexact clone detection in the reference paper consists of: separating the operators/opcodes from the arguments/operands, separating the types of the latter from their value, and replacing the specific value of each with a number indicating how many different arguments have already been used before within the scope of the current window — that is, they are simply numbered in order of occurrence. Now, the concrete number or address used in the original code is no longer included in the data set, but the information that some subsequent instruction uses the same one as a previous one is retained. Also, the combination of the opcode and the first operand type, as well as the combination of the first two operand types are included in the vector describing an instruction. This construction reinforces the idea that these features are particularly important, as substituting for example the operator would change both the opcode feature itself and the one combined with an operand type, but seems arbitrary and might not improve detection results significantly.

Instead of further expanding the format of representation of each instruction using these compound features, I suggest changing the scheme of numbering for the operand values: Each time a value is used (ordered from top to bottom, being the default order of execution, and by arbitrary choice left to right inside instructions), the number of previous occurrences of this value should be recorded instead. The information that is lost is the equivalence of two arguments used inside the window, but when viewing the window as a unordered collection of features (see below) whose incidence is only counted, these couplings are not represented in the resulting vector either way. In return the transformation becomes invariant to shifts in the numbering that occur when some instruction is introduced previously or the order is changed and each relocated instruction's operands are assigned a different number. See for example Figure 4.1, in which registers are loaded with constant arguments for a subsequent function call, but the incidental re-use of one argument causes a significant difference in the numbers of operands in any following instruction inside the window, while anonymous normalization only acknowledges a difference in the location of the argument value used again. In this case, depending on the called function, the two parameter registers have a different semantic meaning and the fact that a `0` is used as a parameter twice likely is not important and the `0`s should actually be considered to be distinct numbers, but there is no clear way to determine that automatically.

Still, only code regions which have a very similar pattern of using values will be described by equal or similar features, the number of arguments which have been used any specific number of times has to be the same in order to imply equality of the feature set.

```
li r4, 0            // 0, 1, 2, 0, 0    // 0, 1, 1, 0, 0
li r5, 4            // 0, 3, 4, 0, 0    // 0, 1, 1, 0, 0
li r6, 1            // 0, 5, 6, 0, 0    // 0, 1, 1, 0, 0
li r7, 2            // 0, 7, 8, 0, 0    // 0, 1, 1, 0, 0

li r4, 0            // 0, 1, 2, 0, 0    // 0, 1, 1, 0, 0
li r5, 0            // 0, 3, 2, 0, 0    // 0, 1, 2, 0, 0
li r6, 1            // 0, 4, 5, 0, 0    // 0, 1, 1, 0, 0
li r7, 2            // 0, 6, 7, 0, 0    // 0, 1, 1, 0, 0
```

Figure 4.1: Excerpts from a pair of windows and their normalized forms: first with the operand order, then anonymously

This way of normalizing the operands of a window will be referred to as "anonymous normalization", and while both the combination features and this method can be activated independently, when evaluating (Chapter 5) the implementation, only the original and the jointly modified settings are considered.

Additionally, since IDAPro recognizes several different addressing modes, and presumably the information which one is used is hardly irrelevant when looking for clones, the number of types distinguished in this implementation have been enabled to go beyond the three basic categories of memory addresses, registers and ("literal" or "immediate") numbers considered in the original paper. The exact number often depends on the state of the disassembly and the way IDAPro processes code of the particular machine in use.

Having counted the operators, operand types, optionally the combinations mentioned above, and the abstracted operands (either in the original way or the one suggested above), for each of these features one dimension of the vector for the whole region is filled with the count of all possible values of the feature in the window. Equivalently, a unordered set — that is, a sorted list which no longer holds information about the original order of the instructions the features arose from — containing the values extracted from the instructions is associated with each feature; this representation is more memory efficient and easier to compare. Concludingly, every but the last *window size* − 1 instructions of any sufficiently long function is the initial point of feature matrix as "wide" as the normalization algorithm produces it and as "high" as the windows taken from the code are. Once again, the feature matrices of all windows from a function are ordered and stored in a dictionary keyed by their position in the function, and each dictionary in turn is the value indexed by the function name in the data structure that results from this algorithm.

4.3 Buckets

	string	file stem	
Parameters	integer	hash vector length	9 (diminishing returns)
	integer	hash value domain	4294967296 (arbitrarily chosen)

As comparing all of the windows generated from across a software system quickly becomes an infeasibly expensive task as soon as the systems analyzed grows above a certain size, they have to be sorted before-hand into buckets of likely mutually similar windows by a suitable hash function. A hash function maps arbitrarily large data to a small, fixed-size hash digest which can be used for efficient storage and comparison, usually by responding to small changes in the input data with drastic changes of the resulting hash value, which also makes them interesting for cryptography applications if some further conditions are met The hash functions applied here are not cryptographic hash functions, rather, they are "continuous" in the sense that they map similar data structures to the same hash value with a certain probability. While some positive hits of the comparison algorithms are inevitably lost, because code regions cut out of functions are not inherently grouped in well-delineated clusters (as maybe the functions are depending on the organizational structure of the system), any desired balance between running time and recall can be struck using a suitable hash function that assigns the windows to wider or more narrow buckets.

An even more sophisticated procedure requires applying multiple independent hash functions, resulting in as many buckets as distinct vectors of hashes have been produced, and then continues to compare each of the contents of one bucket with those of "neighboring" buckets that differ only in a certain number of hash values. The less strict the actual comparison itself is configured, the more tolerance has to be exercised between buckets to reliably reach appropriately closely related windows that nonetheless have been stored apart from each other in separate buckets, but also the higher computational effort is required. Specifically, a number of (truly) cryptographic hash functions iterates through the elements of each feature set and the minimal value is kept, so that a vector of minimal values is associated with a window and it then can be placed in the corresponding bucket. This method of evaluating the similarity of sets or multi-sets has originally been developed by Broder [50] and proven to be, assuming independent hash functions, an unbiased estimator of the ratio of elements shared between two sets to the total number of different elements in them.

The more hash values are created for each object, the more precise is the estimate of set similarity, but also the more values have to be checked for determining "neighboring" buckets. It does not make sense to let the hash vector approach the size of the full data object, in this case the feature sets of a code window, because then there would be almost all of the information already encoded in the hash and therefore most buckets would only include one window (as any slightly different window returns a different hash value in at least one of the functions). The search for related buckets with the contents of which a bucket should be joined for potential clone candidates, then, would be equivalent and with these large hash keys as laborious as looking for related window feature sets directly. In the light of the diminishing returns that a high number of hash functions provide, I chose an arbitrary number of 9, in order to keep buckets at a manageable size and also allow for a gentle relaxation of the partition in buckets by allowing "neighbor" buckets that are still mostly similar in value. Compressing the information contained in $11 * \textit{window size}$ feature values in a comparable number of hash values would defeat the purpose of hashing by dramatically shrinking the possibility that different feature sets

map to the same buckets.

In this thesis, a very rudimentary approach of hashing was taken, by using the local Python implementation's inbuilt pseudo-random number generator. On each iteration, the generator is seeded with the value of the element taken from the set and the first n random integers it produces are compared with the previous minima, where n is the size of the resulting hash vector. That way, sets that contain largely the same numbers result in the same minima in most of the places in the vector. Each of those places can be seen as corresponding to a hash function, and each hash function is implicitly generating a permutation of all possible values that occur in the feature sets. The process is deterministic (within the concrete Python implementation) and therefore returns the same values every time, which makes the hashing results reproducible, instead of slightly changing the sorting into buckets because certain values that were present before but not producing any deciding minimal values now do appear early in the permutation according to one of the hash functions.

An immutable string equivalent of the list of hash values that define a bucket is used as its key; the bucket contains windows — sorted by the function they belong to — the same way as the complete database of windows from the previous step. Naturally, every bucket only contains a small subset of all windows, but organizing them according to the original function names sometimes already makes certain tendencies visible, as being assigned the same hash values and therefore the same bucket is a strong indication that a pair of windows will in fact be detected as a sufficiently similar clone in the following.

4.4 Clones

Parameters

string	file stem	
boolean	find similarity inside functions ("autocorrelation")	False, not used in this thesis
boolean	similar regions inside functions may not overlap	True, not used in this thesis
boolean	only compare between versions, not within them	False, not used in this thesis
integer	ratio of differing hash values tolerated	0, except: see Chapter 5
integer	ratio of differing feature values tolerated	0, except: see Chapter 5
integer	minimum clone length	0, not used in this thesis

The buckets are gone through one at a time, and clones are found within the windows of one bucket and any "neighboring" buckets whose key diverges from the original bucket's key only in few values as explained above (the maximum ratio of these diverging hash values can be specified as a parameter). The feature sets of each pair of two windows in such a group of buckets are compared value by value, and again, the potential pair is discarded as soon as a certain threshold of unequal features is exceeded, otherwise it is included in the original bucket's list of clones. Since the sets are already sorted by ascending value, to compare their contents the algorithm has to manage two iterators over them. Whenever a mismatch occurs, the count of mismatches is incremented and the iterator which has encountered the smaller value (instead of both) is not advanced to the next value; this way any future matches are still recognized which are out of alignment

between the sets until the value corresponding to that smaller one is found later by the other iterator, recorded as a mismatch and skipped. The limit of diverging feature values is another parameter to this step of the algorithm.

Apart from comparison thresholds, the user is offered several options to turn off and on, including one for reverse engineers specifically interested in replicated code regions within single functions, which are normally not covered in the analysis. When similar instructions are frequently repeated at a distance equal to the window size (especially when sequences of instructions just as long as a window or a clean fraction of the window size appear many times in some functions) as in Figure 4.2, matching adjacent windows can produce excessively many clone pairs that require lots of memory space and slow down the computation.

Merging (as detailed in Section 4.4) them into larger clone instances again reduces the amount of data appropriately, but first amassing all possible pairs and then discarding or merging most of them is unnecessary overhead when it is clear that repeated regions will be captured as non-overlapping windows. In this case, another option can be activated that requires pairs of individual windows to not overlap to be compared, but which also leads to inconsistencies with respect to the window size: With a small window size, the algorithm finds some overlapping duplication by aggregation which would have been ruled out with larger windows to begin with.

Another option restricts the comparison to pairs of functions of which exactly one has been marked (in the actual implementation, by the suffix `_old` to its name, see Section 4.1.1) as belonging to an alternative version of a program. Version comparison becomes easy, as the "aggregator" described in Section 4.5, which reports which function pairs contain a certain percentage of cloned regions with each other, reports only the pairs that do *not* share a hundred percent of their instructions instead if this clone detection procedure is currently set to version comparison mode.

In Chapter 5 I explain why I considered using, but ultimately did not use a minimum bound on the length (given as the number of windows in the region, so the actual length in instructions depends on the chosen window size) to be able to operate with minimal window sizes and later filter out the resulted cloned regions which have not accumulated to a size where they are likely interesting and carry meaning.

The resulting window clones are no longer stored in buckets, but arranged by the two functions they come from, forming a dictionary of dictionaries of clones. To make it easier to look for specific matches in the resulting data sets, clones are included in both directions, that is both in the nested dictionary value keyed by the names of function f first and then function g *and* in the one keyed by function g and function f inside there. Each clone, already implying the functions by its location, contains only the place the individual windows were taken from in their respective functions.

Merging In practice, a huge number of cloned windows may be found for two highly similar functions, which can be more concisely expressed as cloned regions longer than individual windows are. The original paper by Sæbjørnsen et alii describes how two pairs of windows from two functions f and g can be joined as long as the windows from either

```
stwu        r1        -0x10(r1)
lis         r4        SUIMCR@h # SUIMCR
addi        r4        r4        SUIMCR@l # SUIMCR
lhz         r11       TGCR@l(r4) # TGCR
sth         r11       0x10+var_8(r1)
cmpwi       cr0       r3        0
beq         lt        loc_4007B2C
cmpwi       cr0       r3        1
beq         lt        loc_4007B40
cmpwi       cr0       r3        2
beq         lt        loc_4007B54
cmpwi       cr0       r3        3
beq         lt        loc_4007B68
b         loc_4007B78
li          r12       0
lhz         r0        0x10+var_8(r1)
insrwi      r0        r12       1,31
sth         r0        0x10+var_8(r1)
b         loc_4007B78
li          r12       0
lhz         r0        0x10+var_8(r1)
insrwi      r0        r12       1,27
sth         r0        0x10+var_8(r1)
b         loc_4007B78
li          r12       0
lhz         r0        0x10+var_8(r1)
insrwi      r0        r12       1,23
sth         r0        0x10+var_8(r1)
b         loc_4007B78
li          r12       0
lhz         r0        0x10+var_8(r1)
insrwi      r0        r12       1,19
sth         r0        0x10+var_8(r1)
lhz         r12       0x10+var_8(r1)
sth         r12       TGCR@l(r4) # TGCR
addi        r1        r1        0x10
b         lt        lr
```

Figure 4.2: A complete function from the subject system, disassembled, which repeats short sequences of instructions

function are directly adjacent to each other, whereby large cloned regions of code inside a function are identified and the data set is greatly reduced.

However, there can also arise cross-matching windows: Not only do two windows and their immediate successors match, but also each of them with the successor of the other, and while the fact that these also match strongly suggests that the windows in question from each of these functions also likely match and the code of the functions repeats itself, it is in the interest of culling the result set — down to a size a human can be expected to deal with — that such clones should be eliminated. They can be identified after grouping together the lines in the usual manner of Sæbjørnsen et alii as a clone whose regions are completely contained (these regions or clones will be referred to as "containing" and "contained", respectively) in those of the aggregate clone and is therefore taken out of the set. The replication in the small scale of the regions in either function will likely be noticed by an engineer inspecting the clone detection results, if it is relevant. The merging process is carried out dynamically to prevent unnecessary space consumption and also to speed up the algorithm by only iterating over already optimized clone representations.

It may still be useful to omit this step to speed up the algorithm, because especially with long functions, and many fragments in them that resemble fragments of other functions, the number of checks (for adjacent clones with which to merge, as well as both superfluous "contained" clones and "containing" clones that make the clone currently in question superfluous) required to integrate all of the clones into the list could run out of hand. As long as only the filtered function pairs that are reported at the end of the next phase of this algorithm are of interest, merging clones together is not necessary.

4.5 Pairs

Parameters

string	file stem	
integer	minimum threshold of clone instruction coverage	0.33

Each pair of functions now needs to be assessed by a single scalar value that represents the degree of similarity between them well. As clones between them, even after conducting the merging step, can overlap in one function if the corresponding regions in the other function are apart from each other, simply adding up the coverage of all clones for each function respectively divided by its length would often return values exceeding one. In this scenario, it would be hard to reason about the minimum cumulative coverage required to consider the pair as sufficiently similar to be interesting to the reverse engineer. Certainly, it can not be larger than 1 in order to detect exact copies of functions that either exhibit no cross-clones (explained in Section 4.4) or had those merged into the complete clone region. But even when setting it to a moderate value of for example 0.5, a function f that repeats one short conglomeration of instructions several times (see also Figure 4.2), which appears only once inside another function g, can already amass that amount of coverage even for g to match it. Now, if that short region is common in the system's code, f is reported as similar to a great number of probably unrelated functions, while two other functions that have an almost identical first half do not pass the threshold.

The quality of results improves when regions that are cloned multiple times in different

places (within the other function of the pair) are not counted twice. If however these asymmetrical clones are of particular interest or are known to occur more frequent than the algorithm detects, it could be useful to set two separate thresholds, of which only one function in a pair has to exceed the higher one. For evaluating this implementation, a single threshold of 0.33 was chosen, since in all but the most extreme configurations this value was considerably larger than the average percentage of clones any arbitrary pair of functions was found to have. Furthermore, while the first system whose analysis this thesis is intended to assist features some functions such as f in the above example, few of them repeat a section of their code more than two or three times. A value of 0.33 will catch a function that simply performs a task thrice in succession and associate it with others that also contain the piece of code that executes that task, and the likelihood that functions are semantically related with less than a third of each matching some part of the other is presumably quite low.

When in doubt, it is likely better to set this threshold lower than intuition would suggest most cases of actual similarity between functions will exceed, because the precise value is still kept throughout the process — so the option to test the validity of reported clones more rigorously or even completely ignore them with low coverage ratios is still open after inspecting the results filtered relatively permissively and finding a high number of low quality clones with a low clone coverage ratio. The inclusion of an asymmetric threshold as mentioned above could also be decided on based on the distribution and quality of a preliminary result set.

The final database of clones then, which will probably be sufficient for most of the use cases of the clone detection algorithm, is again constructed as a dictionary of "first function" keys and inside its values dictionaries with keys representing the "second function" of a clone (of course, by an arbitrary order) and values being two percentages denoting the degree of similarity between the two functions.

4.6 Output

Parameters

string file stem

In addition to the structured output in a file, the resulting cloning ratios can be introduced back into the IDAPro database the instructions were taken from. Again, using IDAPython, the disassembly comments at the beginning of each function are accessed and for every clone that includes a particular function, its comment is augmented by a line indicating the other function of the pair and the degree of duplication that was determined before. Including the individual clone regions either in that comment or even displaying their limits inside the actual instruction sequence below would in most cases add a undesirably large amount of lines to the disassembly and make it harder to overlook, navigate and comprehend. Details about which parts of the function were considered similar by the algorithm can be accessed in the JSON files by search.

As the (intermediary) results stored alongside the disassembly database are open to

inspection and can be parsed once again and prepared for presentation in any desired way, executing this script in IDAPro is optional and just a demonstration of how the clone detection can be integrated into the workflow of a reverse engineer. Also, whenever the extracted assembly from multiple IDAPro files is used together, as detailed in Section 4.1.1, this solution becomes less elegant. Simply placing all of the "clone comments" before a function in one of the files, not disclosing if the matching function is in the same file or if not, in which of the others, could create confusion. As is the case with the method of joining described before, this procedure could be well improved to suit the scenario of working on many distinct disassembly files.

5 Evaluation

Now, it needs to be determined whether the results produced by the proposed algorithm are useful in the context of analyzing software systems, and if they are presented in a format suitable to interpret them as a reverse engineer to come to conclusions that were otherwise out of reach or much more laborious to discover. Generally, the result set can be assessed by its precision and its recall, or equivalently its rate of false positives and false negatives. This thesis could not hope to cover all possible configurations of the system in terms of parameters supplied to the algorithm, but nevertheless a proof of concept is provided by the successful application of one sensible parameter set. The improvements suggested in Section 4.2 concerning the "normalization" of assembly instructions are evaluated by highlighting the differences between the reports generated by either version of the clone detector. Also, a general overview is given over what changes to what parameters would influence the results in a certain way.

5.1 Precision and Recall

In order to confirm that the results the algorithm produced were significant, comprehensive and not diluted by excessive false matches that do not reflect actual similarities between functions, I selected moderate parameters that seemed suitable for a first test run.

While including neighboring buckets (by lowering the corresponding parameter, the "inverse ratio of differing feature values tolerated" as introduced in Section 4.4) is promising, I opted to not tolerate any differences in the hash values at first. If the results already are of a low quality, then loosening the algorithm even further would probably not bring about the desired change, although it is possible that some important groups of clones are left out in the strict version of the algorithm. Preliminary runs during the development of the implementation showed that sufficiently and fairly large numbers of clones would be already output by the program this way. Also, the tolerance when comparing actual code regions with each other was also kept at zero for the preliminary runs and the assessment of precision and recall. Later, in Section 5.3, the possible gains of loosening the comparison step are discussed.

Following the reasoning in Section 4.5, and to make different result sets comparable, the minimum coverage for both functions of a match is the same (at 0.33) in all evaluation experiments, as is the number of hash functions used for sorting windows in buckets.

I chose a relatively small window size of 16 instructions, since most of the functions in the analyzed system did not exceed the length of a hundred instructions, and while assembly code is low in information density (see Section 2.2), sections of about ten to 20 instructions are well capable of performing complex computations and bearing an independent semantical meaning. It could well be the case that some changes in the instructions surrounding a clone within a function would make much of the duplicated functionality undetectable for a more coarsely configured detection algorithm. The al-

gorithm presented in the original paper [48] is aimed more towards high-level clones and wholly duplicated functions or even larger parts of a system, and consequently uses large window sizes from 40 up to 500, and the paper never mentions disadvantages of longer windows except for impaired coverage of the complete code.

5.1.1 Precision

Precision, being the ratio of correctly identified clones among the total set of reported clones, can be measured by taking a sample out of the result set, in the case that it would be infeasibly time-consuming to examine it entirely, and marking every clone in the set or sample as either "sensible" or not. Defining "sensible" is up to the reverse engineer here, and — especially with limited knowledge about the system that is currently being analyzed — he may be able to only give educated guesses on some of the results, and furthermore he is given the opportunity to provide additional qualifying information on each clone. (In the following, he may be referred to as the "oracle").

The clone detection algorithm was run on the system described in Section 2.2.4, with the above parameters, and a random sample of 10% of the function pairs that passed the threshold and were reported in the result file was taken. As mentioned before, the system is effectively divided into two address spaces, that of the bootloader and that of the full software. As especially the bootloader was already studied and amended with names and some comments, it was easier to evaluate the correctness of a given result pair in terms of semantics in the code, while for others it could only be determined if the verdict of the automatic detection is warranted by apparent similarities in the code (including the layout and control flow, which was not directly analyzed). Also, the results were split based on the provenance of the clone pair: clones internal to the bootloader, clones within the controller software and clones between these two subsystems.

Many clones were deemed to be too hard to verify, especially without function names (either from automatic recognition by the disassembler, or provided by the disassembling engineers) indicating what their purpose in the system is, and were tentatively placed in the "unknown" category. However, a large portion of these function pairs are matching completely, so the worst-case scenario that all of them are unrelated to each other and constitute no information gain in the reverse engineering of the subject system is highly unlikely.

Since the results differed only marginally with and without the change from combination features to anonymous normalization (explained in Section 4.2), the former and original version of the algorithm is applied in this test run, and a separate set of clones which appeared in the result set after the modification is sampled in a second evaluation. As the changes to the normalization of code windows only removed some features and removed information content from the operand features, no clones were lost transitioning from the original to the new method.

Interpretation The first five designations in Table 5.1 do not imply that identical functions have been found, but a pair of functions which both belong in the specified category, and thus is helpful in forming a set of all functions belonging to that category. Interrupt

Classification	bootloader	in between	software
floating point library utility		3	
PPC utility	3	4	
IO utility	3		
Interrupt Service Routine	15	2	13
task wrapper			5
wrapper	6	2	
wrapper, only different constants	6		
wrapper, floating point library	1		
identical		7	
identical, C library		1	
extension		3	
sub-cases	2		
only different constants		2	
almost identical		2	
similar, but unknown			2
similar control flow, unrelated function		3	
[unknown]		74	28
TOTAL	36	103	48

Table 5.1: Classification of a tenth of all function pairs based on exact-matching 16-instruction windows without anonymous normalization

Service Routines seem to be either abundant in the code or extremely easy to detect, but the usefulness of the clone detection algorithm becomes clear either way, saving a reverse engineer valuable time finding the sections of code these functions reside in and group them. Renaming them could also be carried out semi-automatically with the support of this algorithm, which suggests a cluster of similar functions for parallel editing.

Also, a lot of functions that were identified as “wrappers” by the oracle were matched in the bootloader. Their absence in the software could be traced back to the fact that most functions there are unknown, and in addition I allowed the multitude of matches between two unknown functions to be skipped. It is doubtful if wrappers form a group of functions that is sensible to consider semantically coherent, but this distinction could still be helpful in reverse engineering and separating all such wrapper functions from more complex functionality already reduces some trivial work that would have to be done otherwise. If these clones are not wanted in the result set, a minimum length of functions can help filter these out, since most wrapper functions are short and call only one other function after supplying it with arguments.

As mentioned earlier, I considered introducing a minimal length of clone regions. For smaller window sizes (with the objective of expanding coverage and performing a more fine-grained analysis at the same time), such a simple mechanism could also increase

the average quality in the result set by discarding small, insignificant clones made up of only a small number of overlapping windows — which would include wrapper functions as described here, while retaining larger cloned fragments whose individual parts follow each other with few gaps. Also, wrapper function that actually call only a single other function could be grouped by or associated with that function, in a rudimentary way incorporating dependency and call hierarchy into the analysis.

Identical functions were detected between the bootloader and the full software version, just where they should occur, and not inside either part of the system. This, once again, attests the clone detection method at least a basic utility, but the number of clones (7 in a 10% sample) in relation to the total of about 1000 functions in each component is not suggesting comprehensiveness at all. One of the identical functions actually was one of those statically linked from the C standard library, and also appears in the recall study 5.1.2.

It should be noted that some functions that have been clearly identified as exact copies of each other did not even nearly possess a similarity measure of 100%, rather, the changes in formatting and data structures that were made only in one part of the system changed the operand types so that several windows did not contain the same feature values. An unfortunate weakness of the "sliding window" algorithm, especially without implementing a tolerated quota of differing features, is that functions with minor modifications in the middle of the instruction sequence carry more weight than those close to the beginning and end of a function, since they affect a larger number of windows. Smaller window sizes lessen the impact of singular changes within an otherwise duplicated sequence of instructions, but would not eliminate the problem.

Some functions were deemed to be extensions of others and often had a significantly asymmetric pair of clone coverage ratios, others represent sub-cases of a certain scenario in control flow that is delegated to one of the functions. Apparently, the algorithm is able to detect some interesting clones between the two "versions" of the system beyond exact duplicates and highly similar wrapper and utility routines. Another considerable subset of the "cross-clones" exhibited only little differences in some values or opcodes used, but a definite semantic relationship could not be immediately determined.

An arbitrary sample of functions in the complete software revealed that judging their semantic similarity and the value of the clones among them is neither efficient nor reliable for the purposes of this evaluation. While some appeared to be similar in their control flow and instructions, the oracle, without prior knowledge about the functions, could only speculate if that indicated a meaningful relationship between them. Essentially, he would perform the kind of analysis that would have been necessary from the outset without assistance of the clone detection algorithm.

Lastly, some pairs of functions across the two system components shared a common structure, even the same number of instructions in between branches and other attributes, but no relation between them could be recognized. The frequency of this case of a "definite miss", however, is low compared to the definite "hits" that the algorithm produced, the false positive rate is satisfactorily low. The fact that, as would be usual, overall most of the clones were found between bootloader and software instead of within each of them makes the approach promising, but the proportions between the three clone sets are still

not what one could expect to obtain with a truly comprehensive, detailed and unbiased search. Such a (comparatively) large bootloader of a piece of software could likely contain considerably more clones than the (mostly) library functions and small utilities that have been found so far.

5.1.2 Recall

Since there is no comprehensive knowledge base or documentation of the system available we can not accurately determine the rate of recall of this algorithm run. In fact, the dimension of the total set of duplicate functions or clusters of functions with a close relationship among them can only be guessed, but as significant parts of the bootloader and some parts of the software have been overlooked by an engineer and provided with meaningful names, one can verify if the functions identified and classified so far are detected, as would be expected. A small report dealing with the solution of a particular problem concerning the subject system lists library functions and system calls that were located in the process of solving that problem, as well as more complex task routines that would be registered with the embedded operation system when executing the software. This list may be biased towards the more easily detected patterns in code, but at least in the case of the system calls ("syscalls"), examining the function responsible for branching out in different cases of syscalls suggests that the list of syscalls that appear in *both* the bootloader and the software is exhaustive.

Classification	C library	system calls	OS tasks
perfect match	16	25	
imperfect match			4
no match	4	1	21
TOTAL	20	26	25
additionally:			
incorrect cross-match		2	
incorrect internal match		2	13
i. i. perfect match			1

Table 5.2: Confirmed matches between the two sections of the systems

Interpretation Figure 5.2 shows the coverage of the clone detector in respect to the set of identified C library functions and OS tasks, as well as the complete list of system calls from the report, and furthermore lists other clone pairs that involve these functions that are not the one-to-one pairs expected based on the information provided by the reverse engineer in the aforementioned report. The C library functions can be assumed to be identical in both systems, and all of them that are in fact present in both systems

match, except for `strchr`, `va_start`, `va_stop` and `epi_r28ff`. This is because all of these functions are less than 16 instructions long and therefore produce not a single window (which would have to be of length 16 as per the parameters set), while the ones that match perfectly well all contain at least that many instructions.

It is also not surprising that most of the syscalls indeed match their respective counterparts, and those only. The fact that two of them probably more or less by accident share a short sequence — little more than 16 in length — of identical opcodes and similar operands, leads to the incorrect clones, where one such function also matches the other syscall function in both the same ("internal") and the respective other section ("cross") of the system. The discrepancies between the two instances of `get_queuing_port_id` are literal operands used in an "inconsistent" way that makes the operand normalizer count the operands in the two functions differently. As explained in Section 4.2 and Figure 4.1, the difference between the two cases can not be completely equalized, as the algorithm does not know if using the same value again actually is meaningful in terms of functionality or in fact a coincidence.

The tasks are not well identified at all by the algorithm. Note that the internal matches only occurred in the full software, and mostly between a groups of tasks that have all been named according to the same scheme, differing only in one number: `task01hz`, `task02hz`, `task04hz`, `task08hz`, `task16hz`, `task32hz`. The fact that these functions are going to be registered as operating system task can be understood as an external circumstance, which, although semantically relevant, does not have to be reflected in the content of the functions. It is obvious now that clone detection algorithms may reliably identify system calls and other elementary components of software systems, comparing code regions is in no way a guarantee of comprehensively capturing all functional similarities in the code of a system.

5.2 Improvements

As mentioned in Section 5.1.1, there are no function pairs that are not found with anonymous normalization that previously would have been found without it. The difference set between the algorithm results following the two normalization variants was sampled and analyzed in a similar manner as the set in the original evaluation of precision in Section 5.1.1. Again, the origin of the matched functions divides the clones in three categories, in addition, the distinction between clones that have only (in most cases, by less than 0.05) increased their similarity measure and those that have actually crossed the minimum of 0.35 (all of which arrived at a value over 0.55) has to be made to assess the utility of this change in the algorithm.

Interpretation The overall ratio of the three categories of the result set resembles the distribution observed in the first analysis (Table 5.1). It is unfortunate that only relatively simple wrapper functions are now detected as being clones, and even a pair of unrelated functions now surpassed the minimum of 33% similarity. There are no complex, less obvious clones that were missed before and now detected by the more lenient version

Classification	bootloader	in between	software
task wrapper			[0](3)
wrapper	[2](3)	[4](1)	[2](0)
unrelated function		[1](0)	
[unknown]		[0](8)	[0](4)
TOTAL	[2](3)	[5](9)	[2](7)

Table 5.3: Classification of a tenth of all function pairs based on exact-matching 16-instruction windows with anonymous normalization
[in brackets]: only detected with anonymous normalization
(in parentheses): score only increased with a.n.

of the algorithm. But, as pointed out in Section 4.2 when deciding on a method of normalizing the instructions, small differences in parameter values can still be present in the anonymously normalized form of the code and the exact matching algorithm used in the test run is still not able to find those pairs of code windows that exhibit these differences. The function pairs whose degree of similarity increased slightly as a result of "relaxing" the normalization step consist most of unknown and more complex functions, but as most of them already had a high percentage of matched code, not much is gained by this change in the algorithm.

5.3 Parameters

There are many parameters which one could change to alter the behavior of the clone detector, and most of them can only be discussed in theory in this thesis and could not have been tested and evaluated in the given time.

Window Size While the stride between windows has been fixed to one (for the reasoning, see Section 4.2), and assessing the merit of alternative solutions remains a problem for future work, the window size is one of the most deciding factors in the clone detection process, as was already demonstrated in the original paper. The division of instruction sequences in functions is the first task the algorithm performs and influences all steps that follow.

Sæbjørnsen et alii [48] already mention one of the disadvantages of using large window sizes, namely that smaller functions are not covered at all by the analysis: Any fragments of code that carry a semantic meaning and are cloned, but smaller than the chosen windows, are hardly ever matched with each other because they have principally completely different lines before and after them. The advantage of large window sizes is pointed out to be a higher quality of the results — even after merging shorter clone sequences from a finer algorithm run with a small window size, but in most common cases there is no reason why individual parts out of large blocks of code should not be

detected as similar to the parts of another block. However, problems could be caused by changes (that persist in the normalized form of the instructions) well distributed in a block of code that affect most of the windows that cover it, as opposed to only a short sequence of windows, which will then not match and in total could reduce the recall rate drastically.

Introducing a minimal clone length will even exacerbate this behavior of the algorithm by filtering out some of the matched fragments between differing instructions in the code and make these "not-exactly-duplicated" long code sequences even less likely to be detected by the algorithm. This additional filter only produces better results when the window size is low enough to permit a lot of insignificantly small clones that do not represent any meaningful unit of functionality. This approach, however, is likely needlessly cumbersome in most cases, where the window size would better just be set to a reasonable minimum value and a required clone length greater than that causes the detector to miss potentially sensible function pairs for the reasons explained before.

Instead of conducting various long-window clone searches and (due to incomplete coverage and recall) combining their finding with those of short-window searches to obtain a complete overview of clones in the system, an adequate mechanism that weakens the impact of small-scale modifications on those windows that include the modified instructions could be employed: Then, smaller window sizes could also reliably detect largely similar functions with frequent small changes and only one uniform clone detection run (with the window size set to the likely minimum length of a sequence of instructions that carries a meaning by itself) could already suffice to provide most substantial results to a reverse engineer.

Hash/Feature "Tolerance" Two other crucial parameters that govern how strictly or loosely the clone detection results will be filtered are the ratios of differing hash and feature values that are tolerated for two windows to still be considered similar and used to gain information about the relation between the functions they are coming from. As the evaluation results presented in Section 5.1 show, a loosening of the comparison step might not even be necessary, and rather, appropriate window sizes and the aforementioned modifications (see the previous Paragraph "Window Size") and a sophisticated analysis of the clone detection results (the possibilities here are justified in Section 6) can already enable a comprehensive high-level analysis of an assembly code base. Nevertheless, it would be interesting to see how far this loosening can be applied without compromising the average quality of the detected clones so far as to make them not usable efficiently by a reverse engineer (too many false positives). A test run with a tolerance of one differing hash value and three differing feature values produced an immensely large number of results, and to finish the process within reasonable time, the algorithm itself already sampled from the totality of possible function pairs and considered only a small fraction of all clones that would be potentially reported.

Table 5.4 shows the results of evaluating an extremely small sample from the results of detecting clones using these small tolerances concerning hash and feature values, to quickly assess the viability of this approach. While still a large number of the detected

Classification	bootloader	in between	software
Interrupt Service Routine		1	
wrapper		1	
unrelated function			1
TOTAL		2	1

Table 5.4: Classification of a thousandth of all function pairs based on inexact-matching 16-instruction windows with anonymous normalization

clones can be classified as belonging to one of the usual categories that were already represented in large numbers before (see Table 5.1), some pairs of functions that are now present in the result set are not in any way related that is apparent to the oracle. This preliminary assessment indicates that the quality of results output by this algorithm already deteriorates considerably as soon as small differences in the normalized windows are tolerated, and perhaps an even further method of abstracting and normalizing assembly code should be developed and used instead of allowing any differences between matching windows regardless of their type. As has been reasoned in Section 3.2, it is always beneficial to use available information about the subject data and specialize the applied algorithm for it.

It is, however, also interesting to note that one function of the pair that was identified as a false positive result had a ratio of similarity with the other function of just barely above the threshold of 0.33 and consequently adapting that minimum as suggested in Section 4.5 would have excluded that clone from the results. Optimizing the similarity threshold, and possibly other post-processing techniques, may be used to sensibly reduce the large quantity of function pairs detected by this version of the algorithm.

6 Future Work and Conclusion

The intent of this thesis was to implement a clone detection algorithm and apply it to a real-world reverse engineering problem that is currently in progress. The underlying method itself was adapted largely unmodified from existing literature, but some practical simplifications were made and some other decisions concerning the realization of the program had to be made and defended. These choices, as well as the parameters that could be tested in the evaluation step were severely restrained by time, as was the evaluation, having been limited to a single subject system to analyze.

However, a solid first attempt at reverse engineering was made on the basis of clone detection results. Even an engineer completely without knowledge of reverse engineering or assembly code would be able to recognize functions with a similar appearance in sequence or presented as a control flow graph, but automating this process is already a considerably complex objective. In combination with the estimations of an experienced human oracle, the merits of this clone detection approach become clear, as simple patterns in code that were located before with considerable effort were now identified by matching sets of similar functions with and among each other.

General Considerations Most of the general shortcomings of the approach should be attributed to what could be called "local" clone detection. The methods considered here are not taking into account the surroundings of a part of the code on a level above the function, which is the largest unit of partition of a program. As a result, even conclusions that could be relatively easily drawn from the call graph of the system, the mere length or cyclomatic complexity of functions, are not taken into account. As was the with the subject system that was used in this evaluation, many systems contain a large number of "wrapper" functions which can not at all be associated with any functionality without the context of the calling and especially the called functions.

A reverse engineer realistically has all of this information at his disposal — the only tools required are a disassembler suitable for decoding functions and jump and branch instructions and adequate presentation methods — and therefore would likely be able to combine the two approaches to deal with complex systems with a large numbers of functions. An automated tool operating on this kind of data would provide insights that would otherwise be too laborious to gain by comparing function lengths individually and trying to spot semantically meaningful clusters and patterns in control flow and call graphs. But a task such as the one automated in this thesis, i.e. comparing functions by their instruction sequence and grouping them by similarity, lends itself more to be executed by a machine than analyses of graphs, it is simply more straightforward to implement and involves less complex data structures and calculations. It may not be necessary at all to automate the analysis of dependencies, call and flow graphs, and other such high-level information, as long as they are adequately presented and aid in the interpretation of the clone detection results given by an algorithm of relatively simple design.

Implementation Details Several improvements in the algorithm and its implementation are still possible and recommended to carry out, as for example the suggestion of using (and evaluating) it while constantly varying the window size and observing the results, and then possibly introducing a minimum clone length to obtain on average more interesting and meaningful function pairs with window sizes larger than a functionally significant fragment of code typically is. Generally, the effects of the various parameters prepared in the algorithm on the result set has not been sufficiently assessed. In addition, a mechanism for equalizing the impact that changes in functions have on its representation in the algorithm (as a set of instruction windows) could be devised in the future: As it currently stands, especially large window sizes lead to an "over-interpretation" of small differences in the midst of functions on the side of the automated clone detector. A similar effect — which however favors large window sizes — should also be mitigated (see Section 5.3).

There are reasonable grounds for suspecting that a large number of clones was missed by the algorithm so far, based on the numbers that the evaluation run produced and the size of the complete system. It is unknown if they belong to a certain category of clones, involve chiefly functions whose disassembly has not yet been inspected by a reverse engineer, or if the results produced so far constitute a representative sample out of all the function pairs that could be considered significantly similar and the detection program failed to recognize a large fraction of those pairs. The execution of a "non-exact matching" algorithm that compares similarly hashed windows in addition to those sorted in the exact same bucket should be accelerated considerably to obtain a lot of results that were missed based on relatively insignificant differences in the disassembly and extensively analyze and evaluate these results. One way to achieve this would be to severely reduce the size of the set of windows avaliable to compare and match, which more than proportionally shrinks the number of comparisons that are carried out, by using a stride between the windows of one function of more than one instruction, as explained in Section 4.2.

Another important development would be user interfaces that either work more closely together with IDAPro to integrate into the workflow of a reverse engineer, or provide their own functionality in order to facilitate comprehension of the result sets of the algorithm and visualize them in a useful way. A next step would be using this tool in a setting where even less is known about the subject system than was known about this system beforehand, in order to experience how much and in which way detecting clones can assist in the reverse engineering of a system from the ground up.

Overall, the results support the notion that searching for duplicated and similar segments of code yields useful results for grouping large numbers of functions according to their functionality, although a reliably high rate of recall is difficult to achieve and verify. A step towards broadly establishing application of clone detection tools in reverse engineering has been made, and in a case study, many of the strengths and weaknesses of a selected algorithm have been identified and demonstrated, and incentives for future work have been highlighted.

Bibliography

[1] E. Chikofsky and I. Cross, J.H., "Reverse engineering and design recovery: a taxonomy," *Software, IEEE*, vol. 7, no. 1, pp. 13–17, Jan 1990.

[2] E. Eilam, *Reversing: Secrets of Reverse Engineering*. Wiley, 2008. [Online]. Available: http://books.google.de/books?id=uW-NWabqy3YC

[3] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C); Instruction Set Reference, A-Z." [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[4] Apple, "OS X ABI Function Call Guide: 32-bit PowerPC Function Calling Conventions," 2010. [Online]. Available: https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/LowLevelABI/100-32-bit_PowerPC_Function_Calling_Conventions/32bitPowerPC.html

[5] IBM, "The PowerPC Compiler Writer's Guide." [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF7785256996007558C6/$file/cwg.pdf

[6] Motorola, "PowerPC Microprocessor Family: The Programming Environments For 32-Bit Microprocessors." [Online]. Available: http://www.cebix.net/downloads/bebox/pem32b.pdf

[7] Hex-Rays, "IDA - About." [Online]. Available: https://www.hex-rays.com/products/ida/index.shtml

[8] C. May, E. Silha, R. Simpson, H. Warren, and C. International Business Machines, Inc., Eds., *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994.

[9] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[10] D. Reniers, L. Voinea, O. Ersoy, and A. Telea, "The Solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product ," *Science of Computer Programming*, vol. 79, no. 0, pp. 224–240, 2014, experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016764231200086X

[11] R. Koschke, I. D. Baxter, M. Conradt, and J. R. Cordy, "Software Clone Management Towards Industrial Application (Dagstuhl Seminar 12071)," *Dagstuhl Reports*, vol. 2, no. 2, pp. 21–57, 2012. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2012/3477

[12] S. Bellon, "Vergleich von Techniken zur Erkennung duplizierten Quellcodes," Diploma Thesis, No. 1998, University of Stuttgart (Germany), Institute for Software Technology, September 2002.

[13] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *Software Engineering, IEEE Transactions on*, vol. 33, no. 9, pp. 577–591, Sept 2007.

[14] R. Koschke, "Survey of Research on Software Clones," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no. 06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/962/

[15] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," Tech. Rep., 2007.

[16] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach ," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009, special Issue on Program Comprehension (ICPC 2008). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642309000367

[17] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review ," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584913000323

[18] C. K. Roy and J. R. Cordy, "Towards a Mutation-based Automatic Framework for Evaluating Code Clone Detection Tools," in *Proceedings of the 2008 C3S2E Conference*, ser. C3S2E '08. New York, NY, USA: ACM, 2008, pp. 137–140. [Online]. Available: http://doi.acm.org/10.1145/1370256.1370279

[19] C. Roy and J. Cordy, "A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, April 2009, pp. 157–166.

[20] E. Tempero, "Towards a curated collection of code clones," in *Software Clones (IWSC), 2013 7th International Workshop on*, May 2013, pp. 53–59.

[21] R. Tairas, "Code Clones Literature," Dec. 2013. [Online]. Available: http://students.cis.uab.edu/tairasr/clones/literature/

[22] M. R. Farhadi, "Assembly Code Clone Detection for Malware Binaries," Master's thesis, Concordia University, April 2013. [Online]. Available: http://spectrum.library.concordia.ca/977131/

[23] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding Software License Violations Through Binary Code Clone Detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985453

[24] C. Kapser and M. W. Godfrey, "Aiding Comprehension of Cloning Through Categorization," in *IWPSE*. IEEE Computer Society, 2004, pp. 85–94. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.2693

[25] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K.-i. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, 2002, pp. 87–94.

[26] R. Fanta and V. Rajlich, "Removing clones from the code," *Journal of Software Maintenance*, vol. 11, no. 4, pp. 223–243, 1999.

[27] J. H. Johnson, "Identifying Redundancy in Source Code Using Fingerprints," in *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, ser. CASCON '93. IBM Press, 1993, pp. 171–183. [Online]. Available: http://dl.acm.org/citation.cfm?id=962289.962305

[28] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 109–. [Online]. Available: http://dl.acm.org/citation.cfm?id=519621.853389

[29] TheOpenGroup, "diff." [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/diff.html

[30] E. W. Myers, "An O (ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.

[31] Harris, "Simian - Similarity Analyser | Duplicate Code Detection for the Enterprise." [Online]. Available: http://www.harukizaemon.com/simian/index.html

[32] B. S. Baker and U. Manber, "Deducing Similarities in Java Sources from Bytecodes," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '98. Berkeley, CA, USA: USENIX Association, 1998, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268256.1268271

[33] B. S. Baker, "On Finding Duplication and Near-duplication in Large Software Systems," in *Proceedings of the Second Working Conference on Reverse Engineering*, ser. WCRE '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 86–. [Online]. Available: http://dl.acm.org/citation.cfm?id=832303.836911

[34] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 368–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853341

[35] H. A. Basit, S. J. Puglisi, W. F. Smyth, A. Turpin, and S. Jarzabek, "Efficient Token Based Clone Detection with Flexible Tokenization," in *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, ser. ESEC-FSE companion '07. New York, NY, USA: ACM, 2007, pp. 513–516. [Online]. Available: http://doi.acm.org/10.1145/1295014.1295029

[36] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *J. UCS*, vol. 8, no. 11, p. 1016, 2002.

[37] R. Falke, P. Frenzel, and R. Koschke, "Empirical Evaluation of Clone Detection Using Syntax Suffix Trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, Dec. 2008. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9073-9

[38] J. Krinke, "Identifying similar code with program dependence graphs," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, 2001, pp. 301–309.

[39] zynamics, "zynamics.com - BinDiff." [Online]. Available: http://www.zynamics.com/downloads/bindiff30-manual.zip

[40] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Software Maintenance 1996, Proceedings., International Conference on*, Nov 1996, pp. 244–253.

[41] K. Kontogiannis, R. de Mori, E. Merlo, M. Galler, and M. Bernstein, "Pattern Matching for Clone and Concept Detection," *Autom. Softw. Eng.*, vol. 3, no. 1/2, pp. 77–108, 1996.

[42] N. Davey, P. Barson, S. Field, and R. J. Frank, "The Development of a Software Clone Detector," *International Journal of Applied Software Technology*, vol. 1, no. 3/4, pp. 219–236, 1995.

[43] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1109/TSE.2006.28

[44] A. Marcus and J. I. Maletic, "Identification of High-Level Concept Clones in Source Code," in *Proceedings of the 16th IEEE International Conference on Automated*

Software Engineering, ser. ASE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 107–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872023.872542

[45] F. Lanubile and T. Mallardo, "Finding function clones in Web applications," in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, March 2003, pp. 379–386.

[46] U. of Waterloo : Software Architecture Group, "Assembler clone detector." [Online]. Available: http://www.swag.uwaterloo.ca/acd/

[47] I. Davis and M. Godfrey, "From Whence It Came: Detecting Source Code Clones by Analyzing Assembler," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*, Oct 2010, pp. 242–246.

[48] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting Code Clones in Binary Executables," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572287

[49] Ecma, "The JSON Data Interchange Format." [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[50] A. Broder, "On the resemblance and containment of documents," in *Compression and Complexity of Sequences 1997. Proceedings*, Jun 1997, pp. 21–29.