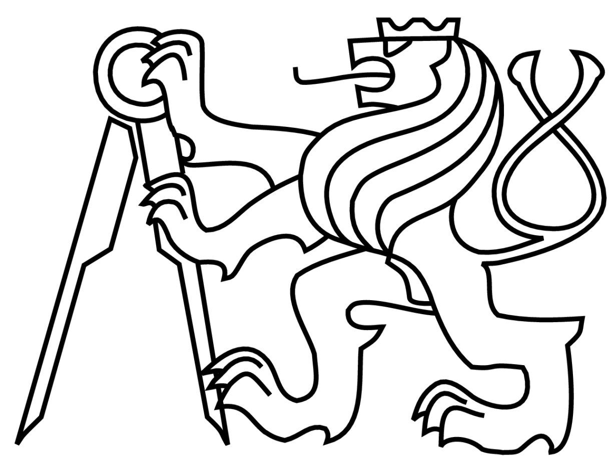
České vysoké učení technické
Fakulta elektrotechnická



Diplomová práce

Systém ověření pravosti
Originality Checking System

Petr Máj

Vedoucí práce: Ing. Ivan Šimeček, PhD

Studijní program: Elektrotechnika a Informatika strukturovaný magisterský
Obor: Systémové Programování

květen 2009

Prohlášení

Prohlašuji, že jsem svou magisterskou práci napsal samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/1200 Sb. o právu autorském, o právech souvisejích s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 15. května 2009

……………………………………

Acknowledgments

Ing. Ivan Šimeček, PhD, *Czech Technical University*
dr Henk Muller, *University of Brisol*

License (BSD License)

Copyright (c) 2009, Petr Máj
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Crosscheck nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY PETR MAJ ''AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL PETR MAJ BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Abstract

This thesis presents Crosscheck, in many ways a novel approach to the source code originality problem. Apart from already available tools Crosscheck utilizes the abstract interpretation and static analysis to combat plagiarized code. It successfully demonstrates both the benefits and drawbacks of this technology and gives a direction to future research into this area.

Abstrakt

Tato diplomová práce popisuje Crosscheck, který je v mnoha směrech zcela novým přístupem k řešení problému ověření originality kódu. Narozdíl od momentálně dostupných systémů, Crosscheck používá abstraktní interpretaci a statickou analýzu k úspěšnému odhalení padělaného zdrojového kódu, úspěšně demonstruje jak výhody tak nevýhody této technologie a otevírá možné cesty pro další výzkum v této oblasti.

Keywords

Plagiarism detection, abstract interpretation, static analysis

Table of Contents

Illustration Index....xiv
Index of Tables....xv
Index of Source Examples....xvi
1 Introduction....1
1.1 Document Organization....1
1.1.1 Typographic conventions....1
1.2 Motivation of the Project....2
1.2.1 Introduction to Source Code Plagiarism....3
2 Background Research....5
2.1 Source Code Plagiarism Detection Tools....5
2.1.1 First Generation Tools....6
2.1.2 Second Generation Tools....8
Jplag....8
Xplag....10
MOSS....11
Other tools....11
2.2 Code Cloning Detection....11
CCFinder....12
2.3 Plagiarism in Human Languages....12
Turnitin....13
2.4 Conclusion....13
3 Solution Overview....15
3.1 Crosscheck's Stages....15
4 Intermediate Language....17
4.1 Crosscheck IL Architecture....18
4.1.1 Memory model....19
Variables....20
4.1.2 xIL Elements....20
Keywords....20
Identifiers....20
Immediates....21
Operators....21
Comments....22
4.1.3 Expressions in xIL....22
4.1.4 Instructions....22
Assert....22

Bind.....23
Call.....23
Eval.....23
Exec.....24
Inparallel.....24
Jump.....24
Return.....24
Sequence 25
4.1.5 xIL Metainstructions.....25
Comment.....25
Control.....26
Function.....28
Language.....28
Line.....28
Name.....29
Source.....29
Type.....30
4.2 Expressiveness of xIL and input languages.....30
4.2.1 Turing Completeness of xIL.....31
4.2.2 Semantic Preservation in xIL.....31
4.3 Translation from C++.....32
4.3.1 Preprocessor.....32
4.3.2 Namespaces.....32
4.3.3 Variables.....33
Unions.....33
Enumerations.....33
Strings.....33
4.3.4 Pointers and References.....33
Operator New.....34
4.3.5 Function calls.....34
4.3.6 Structures and Objects.....34
4.3.7 Control Structures.....35
If and If-Else Clauses.....35
Switch Clause.....36
For and While Cycles.....37
4.3.8 Expressions.....38
5 Abstract Interpretation.....43
5.1 Basic principles.....43
5.2 xIL Abstract Interpretation.....46
6 xIL Code Analysis.....49
6.1 Code Importance.....49
6.2 Code Reachability.....50
6.3 Variable (not constant) Propagation.....51
6.4 Program Flow Analysis.....51
6.5 Abstract Interpretation Specifications.....52
6.5.1 Contexts.....52
6.5.2 Abstract Domain Lattices.....52
6.6 Analyses Example.....56
6.6.1 Simple Example.....56
6.6.2 Interpretation.....58
6.6.3 Variable Propagation Analysis.....59
6.6.4 Reachability Analysis.....60
6.6.5 Program Flow Output.....61
7 Comparison.....63
7.1 Basic Algorithm.....63

7.1.1 Diagonal Analysis....64
7.2 One to One Comparison Strategy....66

8 Evaluation and Results....69

8.1 Crosscheck's Goals....69
8.1.1 Variable and Function Renaming....69
8.1.2 Function Placement Changes....70
8.1.3 Dummy Functions and Variables....70
8.1.4 Extensibility....70
8.1.5 Additional Features....70
Variable Propagation....71
Clever Dummy Code Insertion....71
Statement Reordering....71
8.2 Coursework analysis....72
8.2.1 Task Specification....72
8.2.2 Preliminary Analysis....72
8.2.3 Crosscheck's Results....74
8.2.4 False Positives and Their Explanation....76
8.2.5 Possible Improvements....77

9 Conclusion....79

9.1 Future Developlment....79
9.2 Comments....80

References....81

Appendix A
Attached CD....85

Appendix B
Crosscheck's Brief Tutorial....87

System Requirements....87
Command Line Parameters....87

Illustration Index

Illustration 1: Alignment of the multiplication source and plagiary(1)....................9
Illustration 2: String tiling of example with advanced alterations........................10
Illustration 3: Crosscheck's Architecture Overview..16
Illustration 4: Expression tree..39
Illustration 5: Multiple trees from the single expression......................................41
Illustration 6: Constant Propagation Domain Lattice..44
Illustration 7: Abstract Domains Lattice...52
Illustration 8: Variable Propagation Domain Lattice...55
Illustration 9: Full Program Flow Output at 100th percentile..............................62
Illustration 10: Full Program Flow Output at 90th percentile..............................62
Illustration 11: Reordered Code Match..64
Illustration 12: Diagonal Analysis of Reordered Statements................................65
Illustration 13: Full and Reduced Comparison Visualization................................67
Illustration 14: Mismatching submissions (1,9)..75
Illustration 15: Matching submissions (1,6)..75
Illustration 16: Final Results..76
Illustration 17: CD Contents...85

Index of Tables

Table 1: Levels of program modifications....3
Table 2: Generations of source code plagiarism detectors....6
Table 3: Distance of identifier renaming example....7
Table 4: xIL Keywords....20
Table 5: Precedence of Operators in xIL....21
Table 6: meta control Standardized Values....27
Table 7: xIL Language Values Recognized by Crosscheck 28
Table 8: meta type Families....30
Table 9: Pointer Operations in xIL....33
Table 10: Expression translation rules....40
Table 11: Operator tables for constant propagation....44
Table 12: Code Importance Analysis Rules....50
Table 13: Operator tables....54
Table 14: Variable propagation rules....55
Table 15: C to xIL Variable Names....58
Table 16: Variable Propagation Analysis Results....60
Table 17: Imaginary Variable Analysis Results....60
Table 18: Reachability Analysis Output....61
Table 19: Output Flow Rules....62
Table 20: Preliminary Analysis of the Submissions....73

Index of Source Examples

Text 1: Attribute Counting Example using identifier renaming............................7
Text 2: Multiplication with line boundaries changed..7
Text 3: Multiplication code altered using renaming and dummy code insertion...9
Text 4: Token sequences according to JPlag..9
Text 5: Multiplication - changes to control structure..10
Text 6: Recursive and Iterative version of displayDigits()..................................18
Text 7: meta comment Instruction Example...26
Text 8: meta control Example with two different cycles.....................................26
Text 9: meta function Instruction example...28
Text 10: meta name Example with external calls..29
Text 11: meta line and meta source Instructions Example.................................29
Text 12: Counter Machine Instructions and their xIL equivalents......................31
Text 13: Function and function call example...34
Text 14: Classes and Inheritance in xIL..35
Text 15: Translation of the if-else clause...36
Text 16: Switch clause translation...37
Text 17: For cycle translation into xIL..37
Text 18: While cycle in xIL...37
Text 19: Break and Continue statements...38
Text 20: Simple expression example..39
Text 21: Translated expression in xIL..41
Text 22: Constant propagation example..45
Text 23: Abstract Interpretation...45
Text 24: Sample program after constant propagation...46
Text 25: More complicated constant propagation..46
Text 26: xIL Abstract Interpretation Example..47
Text 27: Code Reachability Example...50
Text 28: Analysis Example - C code...56
Text 29: Translated Analysis Example...57
Text 30: More complicated dummy code..58
Text 31: Variable Analysis in Code...59
Text 32: Clever Dummy Code..71
Text 33: Dummy Code Insertion into the tested submission...............................73
Text 34: Inlining or Extracting Functions..74
Text 35: Recursive false positive...77

1 Introduction

The topic of this thesis was to create a tool capable of detecting plagiarized source code fragments especially in the academic domain. To fulfill this goal I have created *Crosscheck*, the system that is described in greater detail in the following chapters.

1.1 Document Organization

This thesis contains the complete reference to the Crosscheck, notably the explanation of its main design principles. In the first chapter, the structure of the document is presented together with the used typographic conventions which is then followed by a brief motivation into this research area.

Second chapter gives a background research on the currently available tools for the same purpose with their techniques explained as well as an overview of tools sharing remarkable similarities with either the area of plagiarism detection or another Crosscheck's properties.

Third chapter deals with the explanation of the basic idea behind the Crosscheck functionality. Key aspects of the Crosscheck detecting algorithm are then described in greater detail in the subsequent chapters.

Chapters four to seven describe the key principles of the Crosscheck's detection process in their natural order.

Chapter eight evaluates Crosscheck's results and discusses the completion of its goals. The last chapter concludes the thesis and summarizes the work done, and also gives a reasoning about possible future improvements of Crosscheck.

This document also contains several appendices, notably Appendix A describing the contents of the attached CD and Appendix B with brief user tutorial of the whole system.

1.1.1 Typographic conventions

The following conventions are used to simplify reading of the text:

- normal text is typed with a book font
- new items are highlighted in *italics* when they are firstly introduced
- alteration rules and examples are printed on gray background with black border:

```
This is an example how examples are typed.
```

- `identifiers`, **`keywords`** and other `language elements` are typed in monospace font
- source code of any non Crosscheck language is printed with line numbers:

```
1 def exampleCode():
2   print "I am a code example"
3
4 exampleCode()
```

1.2 Motivation of the Project

In over forty years since the first statistics about cheating of university students have appeared, the plagiarism in coursework submissions has grown to one of the biggest educational issues of today's universities.

It is obvious that this problem is not restricted to the academe but presents also a great danger to the whole society, because if students cheat their work in schools they are likely to become incompetent graduates who may fail in their jobs causing damage not only to their employer but also to the potential customer (when the failure is not revealed) and even to the reputation of their university. Moreover cheating also harms the student as he/she is loosing a chance to learn properly thus being unprepared and uncompetitive for future employment. It also harms other students as it constitutes unfair environment [Dick02].

In the recent years the rise of Internet and digital information sharing makes it even easier for students to plagiarize. Previously seen as only lack of original thought, cheating can now be done using the “copy-paste” methods and thus also significantly decreasing the time spent on the particular assignment. Internet searching engines can simplify this task even more by locating the appropriate resources by only a few keywords in virtually no time. There are even specialized web pages - the so called essay mills - offering complete essays and homeworks on common topics for download. Naturally with cheating this easy it is not surprising that recent studies, such as [Sheard02] revealed an overwhelming majority of students (96%) used cheating at least once during their academic career.

Although the answer to the question why students cheat at the first place should not be covered by this thesis (further information can be found in [Dick02] and others), it is worth mentioning at least the basic factors influencing student's decision whether to cheat or not. These are [Dick02]:

- technology
- societal context
- demographic factors
- situational context
- and the personal domain.

Although the prevention, detection and response to cheating in the Czech Republic (and other eastern Europe countries [Grimes04]) is generally milder than in their western counterparts, especially in the English speaking world, the seriousness of such academic dishonesty is well appreciated and appropriate policies are being developed. Such violation at the Czech Technical University usually means a loss of the particular course[1].

Cheating is also not unique to the computer science domain or even engineering in general and sophisticated systems have been developed to deal with the plagiarism of written essays (usually in English), such as the turnitin.com which is described in one of the following chapters.

1.2.1 Introduction to Source Code Plagiarism

Where the main (and the only interesting) tool for essay plagiarism is the paraphrasing without proper citations, the situation in the field of programming languages cheating techniques is much more systematized, mostly due to better formal structure of computer languages.

Various source code plagiarism techniques have been described and categorized by [Faidhi87] to the six levels described in table 1.

Level	Code Alteration Method
0	Original source code
1	Comments and whitespace characters changed
2	Identifier and function names changed
3	Variable positions changed
4	Combinations of functions changed
5	Program statements changed
6	Control logic changed

Table 1: Levels of program modifications

The higher the level the more complex the modifications are and the more time consuming the plagiarism is. Generally speaking the detection of cheated sources where modifications of level four or higher were used is extremely hard. On the other hand such alteration usually requires good understanding of the problem and may be even more time

1 According to the rules a subject may be attempted only twice. If the second attempt is unsuccessful, student is automatically expelled.

consuming than writing the coursework from the scratch. Therefore it is questionable whether such actions still qualify as cheating and these must be dealt with accordingly to the properties of the particular assignment.

The above presented table can be updated to reflect also multi-paradigm programming languages and even cheating across programming languages. The difficulty with these levels is that they do not fit into the hierarchy because they heavily depend on the coursework topic, paradigms used and selected languages. While it may be extremely easy to detect plagiarized C program submitted as C++ application, the same decision with languages such as C++ and Prolog is unsolvable for most cases[2].

Another important factor of source code plagiarism detection is the nature of assignments being checked. Especially in the undergraduate courses, where the majority of cheating is done, due to larger amounts of enrolled students assignments are usually the same for the whole class, as it is unfeasible to prepare unique assignment for each student. Additionally if the whole class receives the same assignment, the submissions could then be compared against each other (for instance in terms of performance) to produce fair and accurate marking.

Although the set of such assignments is vast, most of them are targeted to at least one of the following purposes:

- to demonstrate the knowledge of a particular programming language. These assignments usually take form of a simple algorithmic problem that student usually knows how to solve, the real task is to write the solution in a given programming language
- to demonstrate the knowledge of a particular algorithm domain, which usually means a task which requires students to slightly modify (and/or merge) some of already known algorithms in order to fit them for a particular (usually not trivial) problem.
- performance based assignments. Here students have to understand in great detail some problem and then come up with a solution that is optimal in previously selected characteristics[3].
- optimization based assignments. Although these are very similar to the performance based tests mentioned above, the biggest difference is that these usually do not require program modifications. Rather than that simple local tuning of program's features should be enough to accomplish the goal. For instance in evolutionary computing depending on the operators' probabilities very different results can be obtained. And finding the right probabilities is one example of an optimization based coursework.

From the plagiarism detection's perspective all of the above mentioned assignment types share some trivial similarities, but they are very different with respect to their susceptibility to higher level code alterations, an issue that will be described in greater detail in latter chapters.

2 Putting apart the feasibility and efficiency of such task.
3 For example performance oriented task is to create an assembler program for matrix multiplication optimized for processor cycles required to its completion. This task is given each year to the students of X36APS class at the FEE CTU.

2 Background Research

With plagiarism being such an important problem it is not surprising that Crosscheck is definitely not the first and neither the last tool to detect cheated submissions. The major already existing similar tools are briefly reviewed in this chapter with respect to their relevance for the Crosscheck's intended purpose.

As the topic of source code originality shares some similarities with other fields, notably the natural languages' plagiarism detection and code cloning detection, these are also briefly introduced.

For each reviewed tool an example plagiary is also given to illustrate how different plagiarism detection tools respond to various cheating techniques as well as to introduce these basic code alteration methods.

2.1 Source Code Plagiarism Detection Tools

Programs capable of detecting plagiarized submissions were firstly introduced in the early 80th and during the past 20 years they have evolved enough to be divided into three distinctive generations:

Generation	Description
1	Statistical characteristics (also called attribute counting systems) of a source code, such as number of words, number of lines of code (LOC), parenthesis count, number of variables, etc.
2	Source code is transformed into a simpler form where various irrelevant features of the program are lost, such as whitespace characters, variable and function names, etc. The resulting sources are then matched against each other.

Generation	Description
3	Exploits the semantics of a program at least to some degree. This includes checking code blocks in the order they are executed in a program, changing function calls, etc.

Table 2: Generations of source code plagiarism detectors

It is obvious that the greater the level of a particular tool the wider is the range of revealed plagiarized submissions. However, a threat of over evolved plagiarism detector is very real as all submitted programs are very likely to behave in the same way, i.e. to be semantically equivalent. And with smaller and more precisely defined assignments it is even more probable that two not plagiarized (i.e. clean) programs would be marked as plagiarized because they are semantically equivalent[4].

2.1.1 First Generation Tools

As already briefly mentioned, attribute counting systems (also called feature comparison systems) compute several characteristics of the program source code into a feature vector which is then compared against other vectors using usually standard Eucleidian metrics to determine the distance of the submissions.

According to [Jonas01] these characteristics can be described using the following profiles:

- the physical profile, which consists of properties of the source text that are not related to any specific language. These often include LOC, characters per line, average length of words, character statistics, etc.
- the Halstead profile grouping traits related to individual (programming) languages. This can vary from simple metrics as the average token length, token statistics and tokens per line to more complicated statistical analysis of token sequences
- and the composite profile being the combination of both

The following simple example illustrates both the advantages and weaknesses of the feature comparison system plagiarism detection. Both source codes are used to compute the Fibbonaci numbers, the left one being original, and the right one deliberate plagiate. These submissions are then compared using vector consisting of the following features:

- number of identifiers, keywords, and literals used
- average number of tokens per line
- average length of token
- average number of operators per line

4 This observation is very important as this is one of the key motivations for Crosscheck's ability to tune its precision for different assignments.

Although the most advanced systems from this category consisted of more than 20 different characteristics (e.g. system presented in [Faidhi87]), the above presented list represents a reasonable set of features to demonstrate the possibilities of the method.

```
1 int multiplication(int x,int y) {     1 int mult(int p1,int p2) {
2   int result=0;                       2   int r=0;
3   while (x>0) {                       3   while (p1>0) {
4     result=result+y;                  4     r=r+p2;
5     x=x-1;                            5     p1=p1-1;
6   }                                   6   }
7   return result;                      7   return r;
8 }                                     8 }
```

Text 1: Attribute Counting Example using identifier renaming

It is obvious from the source codes that only the most basic method, i.e.the identifier renaming was used. The feature vectors and their distance are presented in the following table:

Feature	Original	Copy	Second
# of identifiers	11	11	11
# of keywords	6	6	6
# of literals	3	3	3
# of tokens per line (average)	4.88	4.88	9.75
average length of a token	2.18	1.61	1.61
Average number of operators per line	2.38	2.38	4.75
Distance	0.57 / 5.468		

Table 3: Distance of identifier renaming example

The final distance is very small as the only one different feature is the average length of a token. Moreover the other features are remarkably simmilar which leads without any doubt to the conclusion that the submissions are plagiarised.

However making the modifications to the code only slightly more complicated by grouping multiple instructions on the same line, we can easily obtain drastically different vector distance (as shown in the fourth column in the above table):

```
1 int mult(int x,int y) {
3   int r=0; while (x>0) { r=r+y; x=x-1 } return r;
4 }
```

Text 2: Multiplication with line boundaries changed

With this still extremely simple change the overall distance has grown roughly 10 times and the submissions might not be recognized as copies now. It is not hard to imagine other, even more obfuscating, yet very

simple modifications that would change also the remaining features (most obvious being the insertion of a dummy code).

The biggest weakness of the first generation tools was the fact that they utilized only the minimal knowledge (if any) about the input language which causes them to be successful only in cases where the two submissions were nearly identical. Even with the usage of advanced Halstead profile, algorithms belonging to the first generation can be fooled even by code alteration techniques from levels 1 & 2.

On the other hand, due to the fact that these tools view the input source only as a set of characters to be statistically examined, they can be used (to some extent) even to address the human language plagiarism as has been done with the first version of YAP [Wise92].

2.1.2 Second Generation Tools

The answer to the weaknesses of the first generation is to increase the awareness of the input language syntax so that its key elements can be emphasized while others (such as comments) can be discarded for the purposes of the evaluation. Comments and whitespace stripping, identifier renaming and unifying and various forms of hashing are all common features of these algorithms.

For the final comparison various string tiling algorithms such as Running Karp-Rabin, or Greedy String Tiling [Wise93] are generally used. Due to increased time needed for these algorithms to compare each pairs of submissions, several new methods of multi phase comparison have been suggested.

Jplag

Jplag is arguably the most advanced and widespread tool nowadays in active use [Prechelt00]. It utilizes the greedy string tiling algorithm and shares basic principles with another well known program, the third generation of already mentioned tool YAP [Wise96].

Jplag first tokenizes the input sources in a special way where same tokens are assigned different meaning based on their position in the text, e.g. block open at the beginning of a function has a different token than block open after a for cycle for instance. As with almost all tools from this generation, comments and whitespace are omitted completely. Identifiers are all converted into a single identifier token and common language elements, such as imported standard library modules, etc. are discarded too.

After the tokenization, the resulting token streams are then compared using the greedy string tiling algorithm whose results produce the final result. An example of this process is shown in the following paragraphs, where the same simple model program as in the previous chapter has been used. The plagiarized code is the last code from previous chapter updated to include dummy code portions that would clearly fool any first generation tool:

```
1 int multiplication(int x,int y) {
2   int result=0;
3   while (x>0) {
4     result=result+y;
5     x=x-1;
6   }
7   return result;
8 }
```

```
1 int mult(int x,int y) {
3   int r=0; while (x>0) { r=r+y;
    x=x-1 } return r;
4   r=r*3+r-2; return r-1
5 }
```

Text 3: Multiplication code altered using renaming and dummy code insertion

The source codes are now tokenized into token streams which are described in the text below. For identifier, the capital I is used, keywords are transformed to their first bold letters, literals are replaced by the capital L and opening and closing blocks are prefixed with their context, e.g. Function being is F{, while begin is W{. Parentheses, semicolons and other non-essential tokens are omitted too:

```
iIiIiIF{iI=LwI>LW{I=I+II=I-LW}rIF}    iIiIiIF{iI=LwI>LW{I=I+II=I-
                                      LW}rII=I*L+I-LrI-LF}
```

Text 4: Token sequences according to JPlag

Although it may not be obvious to the naked eye, these sequences are remarkably similar, with the only exception being the code included in the plagiary as shown in the figure below (the matching parts have been highlighted in gray):

```
Input1:  iIiIiIF{iI=LwI>LW{I=I+II=I-LW}rI            F}
Input2:  iIiIiIF{iI=LwI>LW{I=I+II=I-LW}rII=I*L+I-LrI-LF}
```

Illustration 1: Alignment of the multiplication source and plagiary(1)

This example can be further expanded with additional functions, statement rearrangement (if possible due to data dependencies), etc. Unfortunately one of the crucial weaknesses of this approach is its inability to properly detect the dummy code. As seen even in the short example above, the inserted dummy code (although being placed after the return statement and therefore completely unnecessary[5]) causes large misalignments, or may even fool the greedy tiling algorithm (no polynomial time algorithm is known for this task [Wise93]) into false smaller matches.

Tiling of tokenized strings is also susceptible to changes in the control structure. Consider for instance a situation in which the while loop in the multiplication example is replaced with a for loop. This is clearly a trivial operation (simple loops, i.e. loops with no internal or external changes to the control variable are easily interchangeable even at the automated level [Schwartzbach03]) and thus is ideal for plagiarism purposes.

Another problematic issue are the expressions themselves – a simple rearrangement obeying operator precedences and other important rules may break the tiling under a reasonable threshold. This idea is based on the fact that while best tiling can be always achieved using only

5 Which in this case can be easily detected by various means of the static analysis.

substrings of length equal to 1, this trivial result does not tell us much about the real problem. Therefore a tiling threshold (usually from 2 to 10) is applied to guide the algorithm to less trivial and more telling results [Prechelt00].

While these techniques generally belong to the upper levels of plagiarism levels (with the notable exception of dummy code insertion) and oversensitivity to these issues may be harmful[6], at least minimal tolerance to these alterations is more than desirable. The following example of the already known multiplication example shows the drastic effects of these alterations on the 2nd generation algorithms:

```
1 int multiplication(int x,int y) {     1 int mult(int x, int y) {
2   int result=0;                        2   int result=0
3   while (x>0) {                        3   for (int i=0;i<x;i++) {
4     result=result+y;                   4     result=result+y
5     x=x-1;                             5   }
6   }                                    6   result=result+5;
7   return result;                       7   return result-5;
8 }                                      8 }
```

Text 5: Multiplication - changes to control structure

Although the above created modifications can indeed be done without understanding the true purpose of the original source code (this is hard to illustrate on such a trivial example though), the two programs will result in a very different token sequences and with a tiling threshold reasonably low (considering the small length of the examples) at at least three tokens, the overall comparison is rather miserable:

```
Input1:  iIiIiIF{iI=LwI>LW{        I=I+I   I=I-LW}rI              F}
Input2:  iIiIiIF{iI=LfiI=LI<II++F{I=I+IF}I=I+L  rI-L              F}
```

Illustration 2: String tiling of example with advanced alterations

This drop from 100% cover of source code to only 63% in such a small code would pass the plagiarism detection and clearly shows the limitations of this approach.

Xplag

While the tools from second generation generally understand the syntax (and sometimes to some extent event the semantics) of the input language(s), it is increasingly harder to add support for new languages. Easy, yet extremely efficient way to solve this limitation is represented by Xplag [Arwin06] which uses the intermediate language[7] of a compiler suite (in this case the GNU compiler suite) on which it then performs the plagiarism detection. Not only does this mean that single application can

6 As already mentioned in previous chapters, large changes of the program statements and control logic usually require more time than the coursework itself and are thus inefficient for possible plagiarists.

7 In this meaning the language in which is represented the inner form after the compiler frontend is finished, which may not even be a language in a human readable sense.

detect plagiarism in a wide range of programming languages (C,C++, Java, Fortran, etc.) but also allows the tool to detect plagiarism across these programming languages as they all share a common intermediate language.

Additionally the translation and various optimizations being done by the compiler suite's frontend may be beneficial to the plagiarism detection process as some of the problems mentioned above (really dummy code, smaller control structure alterations) may be unified during the translation. On the other way the big disadvantage in this approach is that this unification cannot be controlled and is usually not designed to meet plagiarism detection purposes. While some complex optimizations may bring two original codes together, other changes (for instance from procedural to object oriented code) that are easily done by humans without changing the semantics will render the two intermediate representations orders of magnitude different. Nevertheless this approach is one of the most intriguing developments in plagiarism detection in the past decade and can be greatly extended using emerging compiler technologies such as LLVM [Lattner00,Lattner04].

MOSS

MOSS, a.k.a. Measure of Software Similarity is another tool created to fight plagiarism in computer classes submissions at the University of Berkeley, California. The initial part of the algorithm is fairly similar to the Jplag as the input sources are tokenized and stripper off the unnecessary elements. The biggest difference between MOSS and other tools is that apart from pairwise juxtaposition, MOSS creates a k-grams (overlapping sequences of consecutive tokens) which are then hashed to allow a search engine to do the job of finding similar tasks. This allows MOSS to test much larger pools that previously described algorithms. They key algorithm is the winnowing algorithm for fingerprinting documents which is described in [Schleimer03].

Other tools

Other tools dealing with the code plagiarism problems together with more technical details about already mentioned tools can be found in [Maj08].

2.2 Code Cloning Detection

Due to its important commercial implications, code cloning detection systems has been around much longer than plagiarism detection tools and since they share some striking similarities their algorithms can be (and were) easily adapted for the plagiarism detection purposes [Burd02].

Code cloning systems are used to find similar portions of source code in large projects (such as Linux kernel [Zhenmin05]) which are usually caused by improper refactoring or copy-paste style programming. Identified code clones are then reported to the user and should be rewritten in order to achieve correct output code.

CCFinder

CCFinder [Kamiya02] is an industrial quality tool for detection code clones in large source code repositories whose basic functionality (searching for copied code) is remarkably similar to the second generation tools in plagiarism detection. However CCFinder includes some important improvements described by various transformation rules which are executed during the tokenization process.

Rules for C++ language include namespace stripping (std::string to string), template stripping (vector<int> to vector) and others. Their detailed description can be found in [Kamiya00]. The rest of tokenization process is fairly similar to that of Jplag, variables are unified, several unimportant tokens are omitted, etc. Furthermore the authors of CCFinder are aware of the fact that while some code clones are serious, others (although cloned as well) are unimportant and should not be reported at all (such as table initializations).

2.3 Plagiarism in Human Languages

Although a comprehensive review of tools used to combat plagiarism in human languages is out of the scope of this thesis, due to the many similarities they share with the computer languages plagiarism detection, it is imperative to give full introduction also to this subject.

Human language plagiarism is very different from computer languages due to the two main facts:

- syntax of natural languages is often extremely complex and context sensitive, whereas most programming languages fit into the context-free grammars category (sometimes even the LL class). Moreover formalized semantics of natural languages is virtually non-existent (although semantics of certain languages, such as C++, is extremely complex, its subsets can be easily tested and checked which is not possible in context sensitive human languages)
- although the idea of paraphrasing (using thoughts and sentences from unreferenced sources in their original or only slightly modified versions) can be roughly compared to rewriting of a computer program, the main problem of computer languages (e.g. the fact that two submissions to the same task are ideally identical) is not the issue in natural languages where their enormous ambiguity and complexity renders two identical submissions impossible even for fairly small tasks.

As the potential market for natural languages is much larger than the market of computer languages (especially in English) there are well established commercial tools available for this task as well as much larger basin of possible plagiaries with even specialized web pages offering papers to almost any topic, the so called essay mills.

This means that plagiarism detectors has to cross-reference the submitted essays not only with their peers but possibly also with the Internet sources and past submissions[8]. Therefore most of these tools

8 Which caused large legal dispute about the possibly author & copyright law infringements by the testing companies as described in [Foster02, Churchill05]

implement the two phase searching algorithms in which at first searching engine is used to narrows the set of likely-to-be originals of the plagiarized work which are then juxtaposed for final results.

Turnitin

Another property of these systems is usually very detailed presentation of their results and automated submission batch testing. The most widely used system is the turnitin.com which is available both for Universities and High Schools throughout the world [Turnitin07, iParadigms07, Carbone01].

Unfortunately due to the commercial nature of this service only little is known about the actual algorithms. Turnitin uses two phase search and final juxtaposition are returns complete report indicating an overall score of the document as well as detailed information about any problematic occurrence.

Another key ability of turnitin is its cooperation with e-learning suites and clear and simple user interface for both students, and professors.

2.4 Conclusion

Due to the limitations of already existing plagiarism detection tools and the nature of the problem, future tools should improve in their semantic understanding of the checked submissions to allow for more exact results. As the semantic understanding of input languages makes any additions harder, using some form of an intermediate language is generally good idea. Additionally, to allow larger submission databases to be processed either the algorithms must be simplified, or a two-phase search should be implemented. An ideal detection tool should also focus on integration with already existing e-learning suites to improve its practicality.

On the other hand, cross-language plagiarism detection seems to be of minor relevance as this particular technique is not suitable for cheating purposes[9] [Arwin06].

9 However recent development of automatic translation tools between computer languages and generally stronger refactoring techniques may prove the importance of this feature in the future.

3 Solution Overview

This chapter presents the overview of Crosscheck's approach to the plagiarism detection. As a short introduction it does not give any reasons for the used techniques nor does it engage in their explanation, both of which can be found in latter chapters dealing each one with a single stage in the plagiarism detection algorithm presented here.

3.1 Crosscheck's Stages

At the beginning each submission is translated into an intermediate language. This translation does not check for any errors in the submissions and tries always to translate as much of the code as possible as it is assumed that all submissions are valid. After the submission is translated into the intermediate language, several analyses are performed on the program to determine the following properties:

1. A program flow is created which is essentially the original program where all subroutine calls have been replaced with their respective code so that the whole program is only one function. Obviously special cases such as recursion or function pointers must be dealt with separately.
2. Code reachability analysis is performed to find any code that will never be executed. Any findings in this test are immediately reported as suspects since unreachable code is always a sign of plagiarized coursework[10].
3. Code importance analysis attempts to identify the key parts of the program. Knowing these parts can not only add weight to the later comparisons but also aid in determining relevant submissions which should then be juxtaposed one to one. Additionally important variables are also determined as equivalence in these variables may suggest plagiarized setup problem.
4. Variable propagation analysis tries to identify the most important constant variables so they can be used in the plagiarism comparison or explanation.

10 Or extremely bad programming technique, both of which should not pass unnoticed in the academic environment.

When the analyses are finished the program is streamed into a special parallel string of tokens representing the instructions (with some modifications) and these streams are then matched using slightly modified string folding algorithm. This shows parts which are similar (or same) in both submissions and therefore likely to be plagiarized.

After the comparison, a reporter then converts the findings into a human-readable HTML file. The whole process is also summarized in the following illustration:

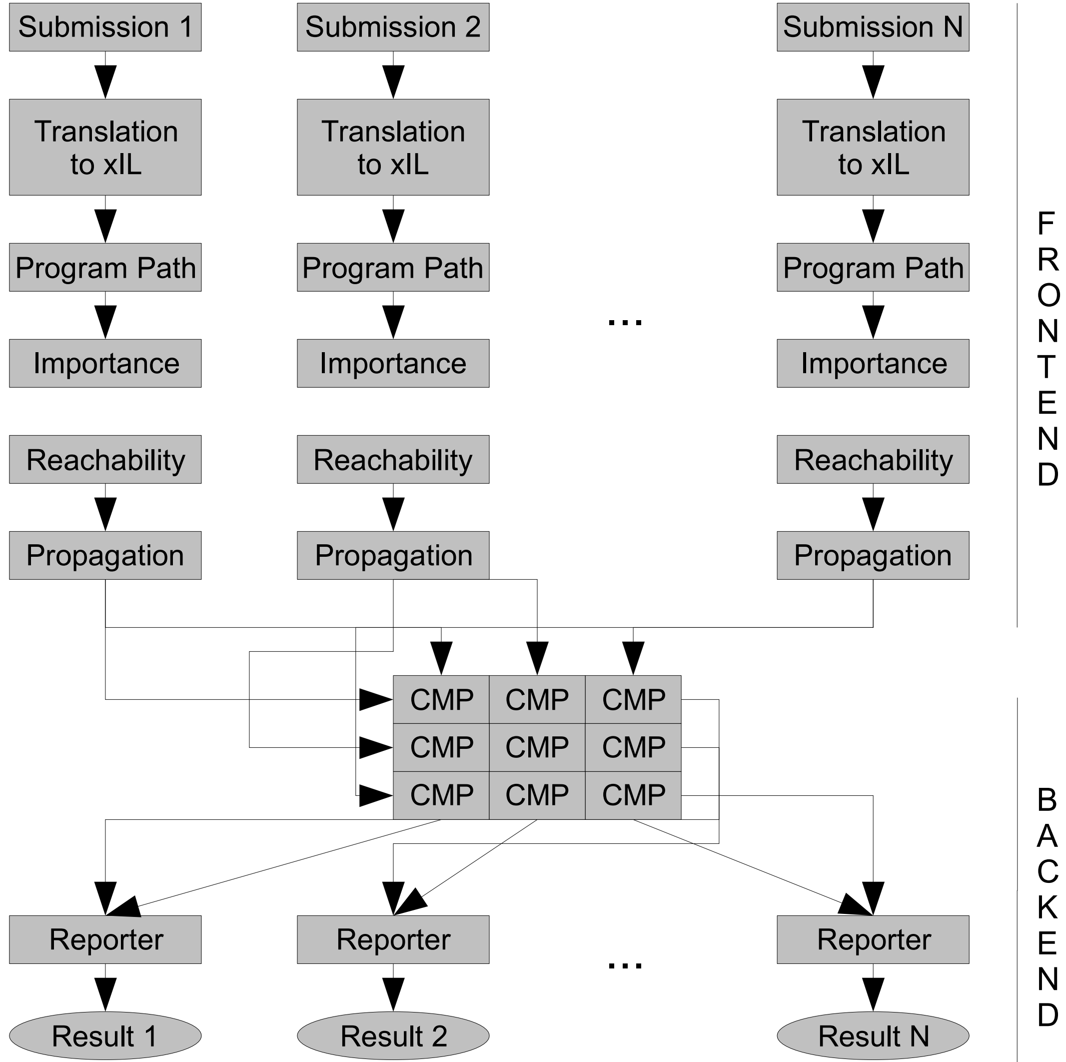


Illustration 3: Crosscheck's Architecture Overview

It is obvious that the most costly part of the process is the actual comparison using diagonal analysis due to the fact that each submission must be compared with any other submission to produce full results.

At the end a careful human inspection of suspected courseworks is (and will always be) required as no automated system can soundly decide whether two programs are plagiarized or only coincidentally similar.

4 Intermediate Language

In order to be capable of detecting plagiarism across multiple programming languages, Crosscheck translates all input sources into the intermediate language (IL). Although this idea of analyzing the intermediate language representation rather than the original code is not new, the previous implementations either used intermediate languages already available as inner form representations in various compiler suites [Arwin06] (notably the GCC and Microsoft .NET), or created rather simplistic intermediate language not capable of preserving many of the original program's features.

The obvious advantage of the former method is that together with the already existing IL the application can also benefit from the compiler suite front-ends (analyzers, parsers) and even code optimizers. However these intermediate languages are designed for a completely different purpose and while they retain 100% semantic similarity with the original code (otherwise translated programs would do something else than sources) they drop most of the information needed to evaluate their similarity for the plagiarism detection purposes, e.g. they lack the information about used control structures, program paradigms, variables, etc. With the usage of code optimizers the issue only deepens: while code optimization may rewrite code statements in a way that semantically equivalent statements look the same way, it may add many false positives to the detection, which is more apparent especially with smaller tasks. Alternatively code transformation done by the compiler suite might add unwanted code (in order to make the program executable) in larger assignments (such as object oriented programming (OOP)) which may later obfuscate the detector.

Let us consider, for example, the simple example of displaying the digits of a given integer in reversed order, i.e. the least significant digit first, the most significant digit last. This task can be easily performed either recursively as shown in the left column, or iteratively (in the right column).

```
1 void displayDigits(int x) {
2   if (x==0) return;
3   cout << x % 10;
4   displayDigits(x / 10);
5 }
```

```
1 void displayDigits(int x) {
2   while (x!=0) {
3     cout << x % 10;
4     x=x/10;
5   }
6 }
```

Text 6: Recursive and Iterative version of displayDigits()

Clearly these two snippets alone are not cheating as rewriting recursion (either way) is more complicated than the computed task in general. However most modern compiler suites will identify the tail recursion present at line 4 in the recursive code and thus rewrite the whole function iteratively which is more efficient in imperative languages[11]. Hence after compiler suite preprocessing these two examples will be false positives, i.e. marked as plagiarized.

On the other hand, it is not hard to imagine two programs that would yield a false negative result. When the algorithm is more complicated, a simple transition from non-OOP to OOP code might add lots of code implementing the OOP functionality which would then lead to smaller proportion of the original code in the compiler's output and thus to overall lower similarity mark. This effect could be even more significant when virtual methods and dynamic typecasting is used.

The second presented option was to design a new intermediate language with keeping in mind the needs of plagiarism detection system. However all previous attempts to do so concentrated on the possibility that when the IL is designed to be simple enough it may introduce significant “optimizations” (in the plagiarism detection perspective) that will drop semantical equivalence with the source in order to simplify the output so that various cheating attempts could be neutralized. Great example of such IL is the inner form used in the CCFinder [Tamiya02].

While such code provides the necessary optimizations for the plagiarism detection, it drops valuable information about the source code structure that can later be used to analyze the outcome and either assist the teacher with understanding the plagiarism methods used, or more importantly determine whether the particular example is cheated or not according to the assignment bias.

4.1 Crosscheck IL Architecture

To overcome the above mentioned problems associated with the intermediate language design, the Crosscheck Intermediate Language (hereinafter only xIL) must fulfill the following features:

- it must be semantically equivalent to any of the possible source languages. Although this can be theoretically achieved by demonstrating that the xIL is Turing-complete (based on the fact that all programming languages are at most Turing-complete), and the xIL indeed is Turing-complete (as is demonstrated in chapter

11 The assembler equivalent of the given examples is not presented as the transition is simplistic.

XXYY), for the plagiarism detection purposes it is logical to assume that the xIL would be capable of mimicking control structures and even paradigms used in the source code. With wide range of source languages it is not possible to satisfy this requirement without the inclusion of meta instructions described later, whose only purpose is to keep information about the used programming style[12]. Another benefit of meta instructions is that plagiarism detection can be computer on classic instructions and when the matches are done they can be filtered using the meta instructions' information.

- The intermediate language must also have as few instructions as possible to simplify the later analysis and these instructions must be flexible enough to be transformed in the later phases of Crosscheck analysis,
- yet it should define certain high-level instructions to prevent the inclusion of redundant implementation code as is done in compiler suites. This particularly leads to a very simple and high level memory model.
- Especially for Crosscheck needs, the xIL must be designed for the latter abstract interpretation. This means the language must support parallelism (as a cut-down version of the nondeterministic computer).
- And of course it will not hurt if the xIL will be at least partially human readable so that the Crosscheck's results can be manually reviewed and understood.

These requirements ultimately lead to a language that shares many common principles with both the assembly language (as the lowest level of human readable language) and some of the modern garbage collected high level languages. Its basic properties are explained and referenced in the following chapters together with various examples of xIL code.

4.1.1 Memory model

Memory in the Crosscheck's IL is garbage collected[13] and invisible to the programmer who can only access the memory using variables. While internally each variable is a pointer, these pointers are not available to the programmer with the only exception of function pointers used in call instructions.

12 It is worth noting that the possible source languages should be limited only to one family of languages when using Crosscheck's cross-language capabilities as transformation from various families (e.g. functional and imperative) may result in extremely different code (not mentioning the fact that rewriting problem from one family to another is by far more complex than understanding its principle and thus is not interesting for plagiarism detector)

13 However the garbage collector is not implemented in Crosscheck as its purpose is not to execute the code.

Variables

Variables in xIL have names in specific format, starting with capital 'V' followed by a number representing the variable identifier itself. If a variable name has to be preserved, the binding of the variable should be preceded by a meta instruction.

```
VAR :: LETTER {0} DECIMALN
DECIMALN :: DIGIT {DIGIT}
DIGIT :: '0' | … | '9'
```

Although it is theoretically possible to use the same variable id for more than one physical variable, such practice is highly discouraged as xIL has no techniques to resolve name conflicts, in which always the closest (in terms of nested sequences) variable will be used.

xIL does not require the variables to be declared prior to their first assignment with rules similar to Python and other dynamic languages: The first occurrence of the variable is alsi its declaration, therefore the first occurrence (in the control-flow sense) must be an assignment. Reading from an unassigned variable should raise an exception.

Each variable can be treated as an array using the index operator '[]'. The same rules that apply for single variables apply also for the indexes. When no index used, the default value (index 0) is returned.

The scope of the variable is always from its first occurrence (declaration) to the end of the sequence in which it was declared. Global variables are variables declared in the main program.

4.1.2 xIL Elements

The xIL syntax is case and whitespace sensitive (thus resembling the Python language syntax in a way) where whitespace is used to identify the sequence of instructions. The other language tokens are: keywords (e.g. instruction names), identifiers (variable names), immediates (numerical, character, or string literals), operators and comments.

Keywords

xIL's set of keywords is very limited and contains only the instruction names, meta instruction classes and two boolean constants as is summarized in the following table:

assert	false	return
bind	inparallel	sequence
call	jump	true
eval	meta	
exec	name	

Table 4: xIL Keywords

Identifiers

```
IDENT :: (LETTER | '_') { LETTER | DIGIT |'_' }
LETTER :: 'a' | … | 'z' | 'A' | … | 'Z' | ...
```

Identifiers in xIL are any alphanumeric literals starting with either a letter (from any language) or an underscore followed by arbitrary number of letters, numbers or underscores.

Immediates

```
IMM :: '”' STRING '”' | NUMBER
STRING :: char | '\\' | '\”' | '\n' | '\t'
NUMBER :: {-} (DECIMALN [ FLOAT ] | HEXN | OCTN | BINN)
FLOAT :: '.' DECIMALN
HEXN :: '0x' HEX { HEX }
OCTN :: '0o' OCT { OCT }
BINN :: '0b' (0|1) { 0|1 }
HEXN :: '0' | … | '9' | 'a' | … | 'f' | 'A' | … | 'F'
OCTN :: '0' | … | '7'
```

Immediates in xIL are very similar to other languages with the following differences:

- character literals are written as strings with length 1, all line endings are converted to the unix style '\n' (0x0A)
- floating point numbers cannot use the exponent notation
- octal numbers have the prefix of '0o' instead of '0' known from C

Operators

xIL supports a reduced range of both binary and unary operators known from other languages. The following table lists all available operators ordered by their precedence (decreasing) with a brief explanation:

Operators	Meaning
(), []	Parentheses, Array access
!	Logical negation, or bitwise complement
*, /, %,**	Multiplication, Division, Modulus and Exponent
+, -	Addition, Subtraction
<=, <, >, >=	Lesser or equal, lesser, greater, greater or equal
==, !=	Equal, not equal
&, |, ^	And, Or, Xor (logical and bitwise)

Table 5: Precedence of Operators in xIL

If an operator can have both logical and bitwise meaning, the correct operator is assigned dynamically based on the variable types. Therefore logical and bitwise variants have the same precedence in xIL.

As xIL does not support assignments in expressions, it also lacks all assignment operator variations well known from languages such as C++ or Java. For the purposes of simplicity xIL also lacks support of operators

which can be expressed by other more general ones (for instance bitwise shift is represented by multiplication or division by the powers of 2).

Division in xIL behaves in the same way as it does in most dynamically typed languages, i.e. division of two integer operands will always return an integer, division with at least one float argument yields float result.

Comments

xIL supports comments in the form known from Python programming language, i.e. No mutli-line comments are allowed and single line comments are prefixed with '#'. By default comments are not generated by xIL generators as comments from source files are translated to meta instructions.

4.1.3 Expressions in xIL

Expressions in xIL (the right hand sides of assignments using the eval instruction) are defined using the terms:

- variable, and immediates are terms
- variable indexed by a term is a term

Terms together build the expressions:

- term is an expression
- negation of an expression is an expression
- two expressions joined with a binary operator are an expression
- (expression) is an expression

4.1.4 Instructions

Crosscheck IL commands, control structures and functions are in general called instructions, of which this chapter serves as a reference. The instructions are ordered alphabetically and each instruction is presented with E-BNF defining its complete grammar, a description and a simple usage (for more details refer to the chapter *Translation from* C++, where are listed more complex examples of IL programs together with their C++ equivalents).

Assert

```
ASSERT :: assert CONDITION
CONDITION :: var REL imm | var REL var
REL :: == | != | <= | >= | < | >
```

The assert instruction allows conditional execution upon the result of the evaluation of the condition. If the condition is true, the code following the assert instruction to the end of the sequence will be executed, if the condition evaluates to false, the following instructions are not executed. The condition can be either true, false, or a comparison of either two variables or variable and an immediate.

Using the assert instruction one can easily write the if control structure well known from other programming languages:

```
1 eval v001 = 56
2 sequence:
3   assert v001 == 56
4   eval v002 = “Condition True”
5   exec (v002) # writeln
```

Bind

```
BIND :: bind var = var
```

The bind instruction is used to reference the right-hand side variable into a left-hand side variable. This means that any change to the left variable will also change the right one (as they share the same memory). Once variable is bound the bond will survive the return instructions. To unbound the variable, variable must be bound to itself.

Variable binding also survives the return statements which means that when a variable is bound inside a sequence and this variable is then returned, the binding survives.

A simple example of the bind instruction follows:

```
 1 eval v001 = 45
 2 bind v002 = v001
 3 eval v002 = 67
 4 assert v001 == 67 # true
```

Call

```
CALL :: call (var|imm) '(' [var {,var }] ')'
        [':' var {,var }]
```

Call instruction is used to call a subroutine (in the IL context a sequence) specified either by a variable or by an immediate address. If the sequence accepts input parameters, they can be defined in the parentheses after the sequence name as variables. If a sequence returns any results their variables must be specified after the colon at the end of parameters list. Please note that all parameters are always passed by their value. If by reference behavior is needed, the following syntax should be used:

```
1 eval v001 = 1
2 call 5(v001):v001
3 return
4 sequence byReference(v002):
5   eval v002 = v002 1 +
6   return v002
```

Eval

```
EVAL :: eval var '=' EXPRESSION
```

Eval is virtually an assignment operator and is the only instruction in the IL that changes the variables' values. Due to this behavior, IL is strictly imperative language and is not capable of implementing source codes from functional languages effectively. However this does not mean

Crosscheck cannot be used to check for plagiarisms in the functional languages because the amount of code added to the functional programs in order to convert them to the IL is almost the same in all submissions.

An example of the eval instruction can be found under the call instruction.

Exec

```
EXEC :: exec '(' [var {,var }] ')' [':' var {,var }]
```

The exec instruction is very similar to the instruction call with the only difference being that exec is calling code out of the scope analyzed by Crosscheck. This usually means either a predefined code similar to all submissions, or a call to source language's standard library, etc. Exec is also always accompanied by meta instruction specifying the call.

Inparallel

```
INPARALLEL :: inparallel ':' INSTRUCTION { INSTRUCTION }
```

The inparallel block is used to denote sequential instructions that can be executed in parallel. A member of inparallel block can also be the sequence instruction as shown in the example below:

```
1 inparallel:
2   exec ()
3   sequence:
4     eval v001 = 1
5     exec ()
```

Jump

```
JUMP :: jump [ imm | var ]
```

The jump instruction is used to jump in the code. The only parameter of the instruction is the immediate or variable with address of the target instruction. Jump cannot point to named sequence instructions with parameters and those which returns any value[14]. Jump is the instruction behind all cycles in the IL:

```
1 eval v001 = 10
2 sequence:
3   assert (v001>0)
4   eval v001 = v001 1 -
5   jump 3
```

Return

```
RETURN :: return [ var {,var} ]
```

The return instruction is used to return from the sequence to the calling code. Return can be followed by any number of variables to return (separated by commas). Please note that in this case as well as it is with

14 This is not checked though.

the function call all variables are always returned by value, never by reference.

The following code shows a simple recursive function that calls itself until its parameter is equal or less than zero:

```
1 sequence (v001):
2   sequence:
3     assert (v001 > 0)
4     eval v001 = v001 1 -
5     call 1(v001)
6   return
```

Sequence

```
SEQUENCE :: sequence [ SEQUENCE_ARGS ]: SEQUENCE_BODY
SEQUENCE_ARGS :: '(' var {, var } ')'
SEQUENCE_BODY :: INSTRUCTION { INSTRUCTION }
```

Sequence is the instruction instantiating a block of sequentially executable code. Sequence instruction can have any number of arguments (expressed as variables) in optional parentheses. Such a block is callable either by call or jump instructions (the latter only if the sequence has no arguments[15]).

Examples of sequence instruction usage can be found in previous instructions.

4.1.5 xIL Metainstructions

Metainstructions are very important concept in the xIL design as they allow the language to express not only the formal semantics of the source program but also the different techniques used. In this chapter the meta instruction and its options is presented in great detail as this instruction is crucial for the Crosscheck's detection capabilities.

This instruction can be anywhere in the xIL source code and it is always applied to the first non-meta instruction after it. If two or more metainstructions are sequenced, they all apply to the first non-meta instruction following them.

Each metainstruction begins with the keyword meta followed by the meta type description and the value delimited by double quotes (double quotes inside the meta value are coded using the escape sequences '\”').

By default, Crosscheck recognizes the following meta instruction types:

Comment

Comment is probably one of the simplest meta types as it is used only to store the comments present in the source inputs. The meta comment instruction is special among the other meta instructions as it actually corresponds to a source input element. Therefore other meta instructions can refer to it as described in the example below, where source and line

15 Note that jump instruction cannot check whether the target sequence returns any values, which may potentially cause a runtime exception.

meta types are referring both to the eval and exec instructions and to the meta comment instruction at line 3:

```
1 # Prints “Hello world”
2 print “Hello world”
```

```
1 meta source “example.py”
2 meta line “1”
3 meta comment “ Prints \”Hello world\””
4 meta line “2”
5 eval v001 = “Hello world”
6 meta name “cout”
7 exec(v001)
```

Text 7: meta comment Instruction Example

Control

Control is a special meta instruction that is used to determine what kind of control structures has been used in the input source. Similarly to the meta type instruction, its values are not determined exactly, however, some standard values used throughout Crosscheck are presented in the following table. Control is particularly helpful when identifying means of plagiarism used in different submissions.

For example, consider the following original and plagiarized pieces of code. In this simplified example, it is obvious even to the naked eye that the cheater changed the original for cycle to while and although the xIL code (without meta instructions is virtually identical in both cases (the situation would be only slightly different if do-while cycle would be used) the control meta instruction clearly preserves the information of how these two blocks of code were translated:

```
1 for (int i=0;i<10;i++)
2   cout << i << endl;
```

```
1 int i=0;
2 while (i<10) {
3   cout << i << endl;
4   i++;
5 }
```

```
 1 meta line “1”
 2 meta control “cycleFor”
 3 eval v001 = 0
 4 sequence:
 5   assert v001 < 10
 6   meta line “2”
 7   eval v002 = “\n”
 8   meta name “cout”
 9   exec (v001,v002)
10   meta line “1”
11   eval v001, v001+1
12   jump 4
```

```
 1 meta line “1”
 2 eval v001 = 0
 3 meta line “2”
 4 meta control “cycleWhileDo”
 5 sequence:
 6   assert v001 < 10
 7   meta line “3”
 8   eval v002 = “\n”
 9   meta name “cout”
10   exec (v001,v002)
11   meta line “4”
12   eval v001, v001+1
13   jump 4
```

Text 8: meta control Example with two different cycles

Value	Description
cycleFor	A cycle which automatically manages its own control variable and is usually executed for a known number of repetitions. A typical example is the for(;;) cycle from the C/C++ languages.
cycleWhileDo	A cycle driven only by the condition that will execute 0, 1, or multiple times, i.e. the condition is evaluated before evaluating the cycle body, e.g. while cycle from C/C++.
cycleDoWhile	Cycle driven by the condition, where the condition is evaluated after the body resulting in the body of cycle being evaluated at least once in each case. Typical example is the repeat – until cycle known from various Pascal language dialects.
cycleForIn	Cycle iterating over an iterable object or collection. Similar to the for cycle because no control variable is used. This cycle is usually found in high level languages, such as Python (for in) and is commonly known as for each cycle.
cycleBreak	Statement used to terminate the execution if the whole cycle from the cycle's body (e.g. a jump to first instruction after the cycle body in assembly language).
cycleContinue	Statement used to terminate the current pass of the cycle, e.g. jump to the cycle's condition.
conditionIf	Standard conditional statement known from almost all programming languages.
conditionElse	The part of a conditional statement that is executed if the condition is false.
conditionCase	Multiple branching either upon an integer variable (C/C++) or general (Python). Usually known as switch statement (notable exception being Python language which uses the elif statement for the same purpose).
conditionIfOp	Ternary operator "?:" known from C/C++ and other languages.
exceptionRaise	Statement used to raise an exception, in general informing the user about an error.
exceptionCatch	Statement catching the exception attempting to remedy the situation.
function	General function.
method	General method, i.e. function with first implicit parameter being the object itself.
methodVirtual	Virtual method

Table 6: meta control Standardized Values

Function

Function meta type determines the definition of a function. The value of the instruction is the name of the function. This meta instruction is followed by couples of meta type and meta name instructions identifying each of the function's arguments (where applicable).

```
1 void myFunction(int i) {
2   int test=i+1;
3   return test
4 }
```

```
1 meta function “myFunction”
2 meta type “integer”
3 meta name “i”
4 sequence(v001):
5   meta type “integer”
6   meta name “test”
7   eval v002 = v001+1
8   return v002
```

Text 9: meta function Instruction example

Language

The language meta instruction determines the language of input source from which the xIL is taken. Along with the line and source metas, language is also valid for multiple following instructions until new language is defined.

As each new Crosscheck's input language should add its own value for the language, in this project only the following language values are used:

Value	Language Description
c	Standard C language (based on the gcc)
cpp	Standard C++ language (based on the g++)
java	Standard Java language (base on the Sun Java 1.4)

Table 7: xIL Language Values Recognized by Crosscheck

Although additional languages are possible, their language identifiers will be ignored by Crosscheck[16].

Line

To determine the line of input source the meta line instruction is used. The scope of meta line instruction is not only the next non-meta instruction, but the line is valid until next meta line or meta source instruction. Therefore if a single input source line is translated into multiple xIL instructions, meta line instruction must be the first instruction in the sequence, as is shown in the example for meta source instruction later in this chapter.

16 This does not mean that Crosscheck will not be able to check if they are plagiarized, only that no additional information will be available on positive matches (assuming no modifications are done to the Crosscheck itself).

Name

This meta type is used to determine the name used in the input source for identifiers (e.g. variables, new types, etc.). For a simple example, see the meta function. Names are not assigned to temporary variables in the input source and to the variables created during the translation to the xIL.

The name meta type is also used to determine which external functions are called using the exec instruction, as shown in the example below:

```
1 cout << “Hello world”
```

```
1 eval v001 = “Hello world”
2 meta name “cout”
3 exec(v001)
```

Text 10: meta name Example with external calls

Source

This meta type is used to describe the input source file from which the following instructions were translated. As the meta line instruction, source does not apply only to the next non-meta instruction, but it's scope is extended till new source instruction is found.

Crosscheck puts source and line instructions before each callable sequence instruction to determine the source file for the whole function[17], as shown in the following example (comment metas have been left out for simplicity reasons):

```
1 // File main.cpp
2 #include “example.h”
2 int main(int argc, char** argv) {
3   displayMe();
4   return 0;
5 }

1 // File example.h
2 void displayMe();

1 // File example.cpp
2 #include “example.h”
3 void displayMe() {
4   cout << “This is me!“ << endl;
5 }
```

```
 1 meta function “displayme”
 2 meta source “example.cpp”
 3 meta line “3”
 4 sequence():
 5   meta line “4”
 6   eval v001 = “This is me!”
 7   eval v002 = “\n”
 8   meta name “cout”
 9   exec(v001,v002)
10 meta function “main”
11 meta type “integer”
12 meta name “argc”
13 meta type “string”
14 meta name “argv”
15 meta source “main.cpp”
16 meta line “2”
17 sequence(v003,v004):
18   meta line “3”
19   call 4()
20   meta line “4”
21   eval v005 = 0
22   return v005
```

Text 11: meta line and meta source Instructions Example

17 However, this approach is not required by the pure xIL definition.

Type

This meta is used to specify the type described in the input source. The following families of input source types are recognized and dealt with according to the table below. As the types are usually replaced with a family identifier, at this point some semantical information is usually lost. This, however, does not pose any threat to the Crosscheck's recognizing abilities due to the fact that for software plagiarism detection it is safe to operate only on type families. In fact it is even desirable that simple change from byte to word[18] should appear as identical code.

Type Family	Value	Comments
any integer, boolean	integer	
any floating point	float	
any string, char	string	
any reference	reference	Type is specified by the target object.
any pointer	pointer	
object	class name	Type is the class name unless the class is a built-in special class for which additional rules apply.

Table 8: meta type Families

4.2 Expressiveness of xIL and input languages

In order to use xIL as the intermediate language in Crosscheck, we need to answer the following two important questions:

1. Is xIL able to express all possible algorithms written in any of possible input languages? (i.e. can any program in the input language be translated into a semantically equivalent program in xIL)
2. Does xIL preserve enough information about the input sources to be useful for the plagiarism detection purposes? (i.e. how well does the comparison of two xIL codes estimate the relation between their sources)

It turns out, that the answer to the first question is very simple using the concept of Turing Machine [Kolar04]:

18 For example in the Object Pascal programming language.

4.2.1 Turing Completeness of xIL

To demonstrate that any program written in any possible input language can be also written in xIL it is only needed to demonstrate that xIL is a Turing complete language, therefore any program (algorithm) capable of being run by a Turing Machine (any program written in most current programming languages) can be also written in the xIL.

The Turing completeness of xIL will be shown by proving that another minimalistic programming language that has been previously demonstrated to be Turing complete. Such a language is for instance the counter machine formalized by Martin Minsky in [Minsky67]. This language has arbitrary number of integer variables and the following three functions:

- INC (r) – z:=z+1
- DEC (r) – z:=z-1
- JZ (r, z) – if (r==0) jump to instruction z, otherwise continue

Now it is easy to show how to emulate these instructions in the xIL. The only problem is that in xIL the jump instruction can jump only to the sequences. Not optimal, yet formally correct solution is to preface each instruction in xIL with a sequence instruction, as is shown in the following demonstration:

`1 INC (r)`	`1 sequence:` `2   eval r = r+1`
`2 DEC (r)`	`1 sequence:` `2   eval r = r-1`

`3 JZ (r, z)`	`1 sequence:`
	`2   assert r == 0`
	`3   jump z`

Text 12: Counter Machine Instructions and their xIL equivalents

Although this definition is formally correct, emulating an universal Turing machine using the above algorithm (and using counter machines) will add exponential overhead. Without formal verification, we will state that xIL is indeed Turing complete because it contains multiple variables and arrays, recursion and conditional branches.

4.2.2 Semantic Preservation in xIL

Unfortunately for the plagiarism detection purposes it is not enough to demonstrate the Turing completeness of the xIL as the proof that xIL can code any thinkable program from the input languages does not imply that such a description is usable for cheating detection. Moreover, the mere fact that xIL is theoretically capable of coding any algorithm does not mean that the translation process is simple, nor that the semantics of the internal parts of the program is preserved. This is due to the fact that Turing Machine is concerned only about the form if the input and output on the tape, not the program internals.

Based on the basic principle of Crosscheck's detection, the most important characteristics of the program are the order in which the instructions are executed (only their first time execution) and their importance. Therefore any translation process should aim to preserve as much as possible of these two key properties.

4.3 Translation from C++

To show the capabilities of the xIL, its limitations and practical implications, a translation from the C++ language (which can be viewed as the most powerful (in terms of expressiveness[19]) among the imperative source languages) is described in this chapter. As some of the aspects of the language are almost identical to the xIL they are not covered, while others such as object oriented programming and complex expressions are described in much greater detail.

All of the following examples should also contain numerous meta instructions which will determine the original processes used. However these instructions have been omitted from the examples in this report as they unnecessarily increase the length of the examples and does not provide any additional information to the translation process.

This chapter should not be taken as an xIL or C++ translation reference. Full specifications of xIL semantics and of the translation process can be found in the source code documentation of the respective functions.

Due to space constraints the complete reference together with additional information on metainstructions and other relevant data can be found only in the source code documentation in respective modules.

4.3.1 Preprocessor

While parsing, all local includes (i.e. double quoted includes) are treated accordingly and the required files are imported. Since xIL output is only one file created from possibly many C++ implementation source files there may be several identifier redeclarations which are reported as a warnings during the translation. All other imports are omitted as it is presumed that non-local files are not parts of the submission.

Preprocessor macros are supported only for the purposes of conditional translation and simple constants, which are translated as new global variables.

4.3.2 Namespaces

xIL does not recognize namespaces as each namespace element is unique as a function address, or constructor's address for objects. Other

19 One can argue that Java or Python are more expressive languages due to their advanced high level features such as garbage collection or even dynamic data types. However as these aspects are native to the xIL itself it is obvious that their translation to xIL will be as straightforward as possible. In this sense the “most powerful” can be also viewed as the most different language (within reasonable set of widespread imperative languages).

namespace elements such as various type definitions and not part of the output xIL. Global variables are unique due to their variable addresses.

4.3.3 Variables

According to the xIL specification all variables are transformed to variables. Each variable can be used as an array with either immediates or variables as indexes. For simple types such as integers and floats this mapping is easy. However, the following types need special consideration:

Unions

As all variables in xIL are untyped, there is no need for unions and therefore unions are treated as simple variables (means their members are discarded).

Enumerations

All enumerations are represented as integer types with their values converted to positive numbers during the translation process.

Strings

Strings and constant char pointers are treated equally for the xIL purposes. As each variable can hold a string literal by itself. Using indexes on the variable will given access to specific characters. Any method of std::string is translated into an exec instruction.

4.3.4 Pointers and References

References can be easily emulated using the bind instructions. Anytime a reference variable is initialized to another value, the instruction bind is used. When variable is bound, the bond is carried across return statements thus allowing xIL to return references from functions.

Pointers on the other hand are a concept completely alien to the xIL notion. Changes to dereferenced pointers are normal variable changes in xIL and changes to addresses are equal to binding variables in the xIL. The following table summarizes the most common pointer operations and their xIL alternatives. In all these cases any variable is a pointer.

C++	xIL
p=q	**bind** p = q
*p=6	**eval** p = 6
p[4]=7	**eval** p[4] = 7
p++	**bind** p = p[1]
p=NULL	**bind** p = r0

Table 9: Pointer Operations in xIL

Operator New

While operator delete is quite easy since xIL is in theory garbage collected language, the new operator poses some serious problems. When a new memory is allocated, xIL destroys any bindings by bounding the variable to itself. This essentially creates a new instance of variable which can then be filled with the appropriate contents.

New arrays of simple types are omitted (there is no need to allocate the memory for arrays since each variable is essentially an unlimited array in itself). More complex types are then allocated using their respective constructors.

4.3.5 Function calls

All functions are translated into xIL as parametrized sequences. All variables are passed on a by value basis (meaning that their values are copied). This implies problems when the parameters are either references, or pointers as changes in their values (in the case of pointers of their dereferenced values) should be visible also in the caller's code.

To overcome this problem, xIL uses the technique known from simulations of functional programming languages [Finkel96]. When function needs pointers and/or references they are passed normally by value, and the number of parameters the function returns is increased by one. When the function returns the parameters these are used to overwrite values in the caller's context as is shown in the following example:

```
1 int fnc(int i, int& j, int* k) {
2   i=5;
3   j=10;
4   k=7;
5   return i
6 }

? q=1
? q=fnc(a,b,c);
```

```
1 sequence(r1,r2,r3):
2   eval r1 = 5
3   eval r2 = 10
4   eval r3 = 7
5   return r1,r2,r3

? eval r8 = 1
? call 1(r5,r6,r7):r8,r6,r7
```

Text 13: Function and function call example.

4.3.6 Structures and Objects

First important thing about xIL and Object Oriented Programming (OOP) is that xIL does not recognize access specifiers (such as private or public) and all members and methods are public[20]. Therefore in the following text structures and classes are treated equally[21] (and referenced as classes only).

Any object is represented by a single variable. Class members are stored into designed indexes and so are method addresses. Each method has

20 This makes sense when realizing that xIL does not need to control whether access rights are not violated, an assumption for any program checked against plagiarism is that the program is working, thus corresponding to all formal requirements.

21 This notion is consistent with the general idea of structures and classes in C++, where structures are essentially classes with default access set to public[Eckel00].

automatically added the first parameter which is the object itself in a way well known from the Python programming language. Normal methods are added to new indexes. Virtual methods are stored in the same index as the methods they are overriding. Each methods returns not only its result type, but also the object itself in the first place.

To create an object a constructor must be called. This constructor is not the same as constructors in C++ as the xIL constructor only fills the addresses of methods in the object variable as shown in the following slightly longer example:

```
1 class A {
2   int x;
3   void setx(int j) {
4     x=j;
5   }
6   virtual int doSomething(int z) {
7     x=z+5;
8     return x;
9   }
10 } ;
11 class B:public A {
12   virtual int doSomething(int z) {
13     return z+x;
14   }
15   int getx() {
16     return x
17   }
18 } ;

? A* x=new B();
? x.setx(5);
? x.doSomething(4);
```

```
 1 sequence:
 2   eval r0[1] = 5
 3   eval r0[2] = 8
 4   return r0
 5 sequence(r1,r2):
 6   eval r1[0] = r2
 7   return r1
 8 sequence(r3,r4):
 9   eval r3[0] = r4+5
10   return r3,r3[0]
11 sequence:
12   call 1():r5
13   eval r5[2] = 16
14   eval r5[3] = 19
15   return r5
16 sequence(r6,r7):
17   eval r8 = r6[0]+r7
18   return r6,r8
19 sequence(r9):
20   return r9,r9[0]

? bind r10 = r10
? call 11(r10)
? call r10[1](r10,5):r10
? call r10[2](r10,4):r10,r
```

Text 14: Classes and Inheritance in xIL

4.3.7 Control Structures

Generally, all control structures known from the C++ language are easily transformed to xIL. This chapter lists the most common of them and shows their counterparts in the xIL. Although the following examples contains only blocked examples, single lined statements can always be transformed as blocks containing only one statement and therefore the presentation is satisfactory.

If and If-Else Clauses

If clause is easily transformed into the xIL using the inparallel instruction. In case of missing else statement, the inparallel instruction has only one branch and can thus theoretically be replaced with sequence instruction. However, for the plagiarism recognition purposes this is not performed. Obviously the ternary operator ?: is only a special case of if-

else structure and is dealt with accordingly. The following example shows the translation template for if-else clause:

```
1 A
2 if (condition) {
3   B
4 } else {
5   C
6 }
7 D
```

```
1 A
2 eval r = condition
3 inparallel:
4   sequence:
5     assert r != 0
6     B
7   sequence:
8     assert r == 0
9     C
10 D
```

Text 15: Translation of the if-else clause

It is worth noting that the translation of the condition itself (which is essentially an expression) might be much more complicated and is covered later in a special subchapter.

Switch Clause

Switch case is very similar to the nested if-else statements. When some case branches does not contain break statements, other statements to the first break or end of the switch statement are attached, as is shown in the following example:

```
 1 A
 2 switch (expr) {
 3   case a:
 4     A
 5     break
 6   case b:
 7     B
 8   case c:
 9     C
10   case d:
11     D
12     break
13   default:
14     E
15 }
16 F
```

```
 1 A
 2 eval r=expr
 3 inparallel:
 4   sequence:
 5     assert r==a
 6     A
 7   sequence:
 8     assert r!=a
 9     inparallel:
10       sequence:
11         assert r==b
12         B
13         C
14         D
15       sequence:
16         assert r!=b
17         inparallel:
18           sequence:
19             assert r==c
20             C
21             D
```

```
22          sequence:
23            assert r!=c
24            inparallel:
25              sequence:
26                assert r==d
27                D
28              sequence:
29                assert r!=d
30                E
31 F
```

Text 16: Switch clause translation

Although this method might seem more lengthy and complex than necessary it provides the switch template in its most general form. The insertion of code (statements C and D in the example) additionally increases any possibilities of matches in altered code with the same functionality by grouping code together without unnecessary xIL code.

For and While Cycles

Cycles are translated straight into xIL using sequences and loops as shown in the following examples for the two most common cycles (for and while). Other special cycles existing in other languages might seem quite different (for instance the foreach cycle in Java or the for in cycle in the Python language), yet they can always be easily transformed to the standard cycles with their bodies intact [Aho06].

```
1 A
2 for (init;cond;iter) {
3   B
4 }
5 C
```

```
1 A
2 init
3 sequence:
4   eval r = cond
5   assert r == 1
6   B
7   iter
8   jump 3
9 C
```

Text 17: For cycle translation into xIL

```
1 A
2 while (cond) {
3   B
4 }
5 C
```

```
1 A
2 sequence:
3   eval r = cond
4   assert r == 1
5   B
6   jump 2
7 C
```

Text 18: While cycle in xIL

The two examples above also show the remarkable similarity between the two cycles in the xIL as both the cycles have same starting and ending instructions. This is the simplest possible example of a key function of xIL which is to bring slightly different subscriptions closer[22].

22 The same functionality as can be seen in tools such as XPlag when utilizing intermediate language of compiler suites. However, this ability of xIL is tailored

The following example shows the break and continue statements and their translation in the “more complex” for cycle:

```
1 for (init;cond;iter) {
2   A
3   continue;
4   B
5   break;
6 }
7 C
```

```
1 init
2 sequence:
3   eval r = cond
4   assert r == 1
5   A
6   jump 9
7   B
8   jump 11
19   iter
10   jump 3
11 C
```

Text 19: Break and Continue statements

4.3.8 Expressions

Possibly the hardest part of translation from C++ to xIL is the translation of various C++ expressions as xIL lacks many possibilities for creating them that C++ takes for granted (notably calling functions and using their results inside expressions).

To construct the expression tree, the simplified C++ expression parsing grammar is used with only limited support of operators precedence as shown in the following example. Please note that not all grammar rules are explained (such as new declarations, etc.) for the simplicity reasons:

```
Assignment :: Logical AssignmentC
AssignmentC :: (=|+=|-=|*=|/=|%=|&=|^=|”|=”|<<=|==>)
               Logical AssignmentC
AssignmentC :: empty
Logical :: Comparison LogicalC
LogicalC :: (“||”|&&|^|&|”|”|!) Comparison LogicalC
LogicalC :: empty
Comparison :: Shift ComparisonC
ComparisonC :: (==|!=|<=|>=|<|>) Shift ComparisonC
ComparisonC :: empty
Shift :: AddSub ShiftC
ShiftC :: (<<|>>) AddSub ShiftC
ShiftC :: empty
AddSub :: MulDivMod AddSubC
AddSubC :: (+|-) MulDivMod AddSubC
AddSubC :: empty
MulDivMod :: Item MulDivModC
MulDivModC :: (*|/|%) Item MulDivModC
MulDivModC :: empty
Item :: {“-”} ( “(“ Assignment “)” | ConstantItem |
        VariableItem [Index])
Index :: “[“ Assignment “]”
ConstantItem :: LITERAL|FLOAT|INTEGER
VariableItem :: [*\&]identifier [FunctionCall]
FunctionCall :: “(“ Assignment { , Assignment } “)”
```

especially for the cheating detection purposes.

During parsing, each C++ expression is parsed into an expression tree. This tree is then scanned recursively from its leaves to the root and certain operations are performed that splits the large C++ tree into smaller expressions (if necessary) to allow their translation into the xIL. During the translation process all calls to functions or methods (thus also calls to user specified operators of complex types) are recognized and replaced by proper function calls.

This is briefly shown in the following illustration and text (all variables except the defined variable x and variable y are objects and these objects have redefined operator +.

```
int x=(3+5*(10-o.x(3,5+y))+2+(o+z)
```

Text 20: Simple expression example.

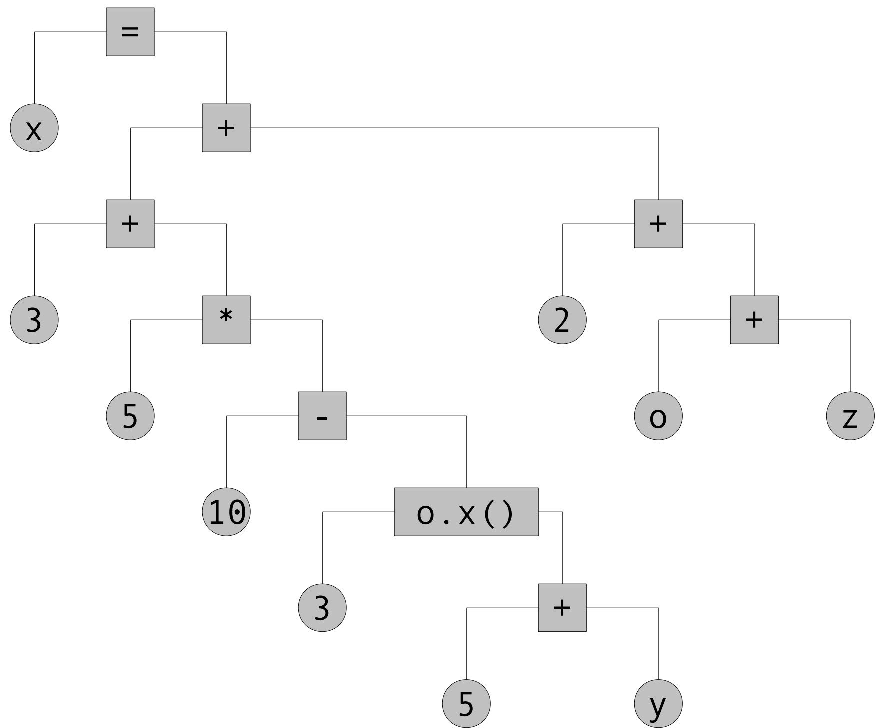


Illustration 4: Expression tree

After the creation of the expression tree, the tree is then parsed and the following rules are checked:

Rule / Root token	**Comments**
Assignment operator	Whenever an assignment operator is found, the whole subtree is then extracted as a new expression which will be evaluated before the processed expression and its occurrence is replaced with the left side of the assignment operator
Function call	When a function call is translated, its nodes can be either simple variables, or expressions. In the latter case these expressions are extracted as assignment operators to new variables which in turn will be used in the function call. The function call is then translated into a call instruction before the parsed expression and is replaced by its returning value (into a new variable) in the expression.
Operator on objects	If operator is defined for the parameters (essentially if the operator is defined for the left operand, the operator is replaced with function call to this operator and the rule for function calls is used.
Special operators on built-in types	When a special operator not available in xIL is found, (such as << or >> (bitshifts)) on built-in variable types is found, it is replaced by its xIL alternative. In this case by integer multiplication or division.
Special operators on complex types	If their codes are not found, they are replaced with an exec call.
C++ operators (++,--) preorder	These operators are evaluated into new eval instructions that are placed after the parsed expression. Their value is replaced by the variable. Additionally these operators are grouped together, e.g. X++++-- is evaluated as eval x = x 1 +

Table 10: Expression translation rules

After applying the above rules, expression tree from the given example breaks into the following smaller trees:

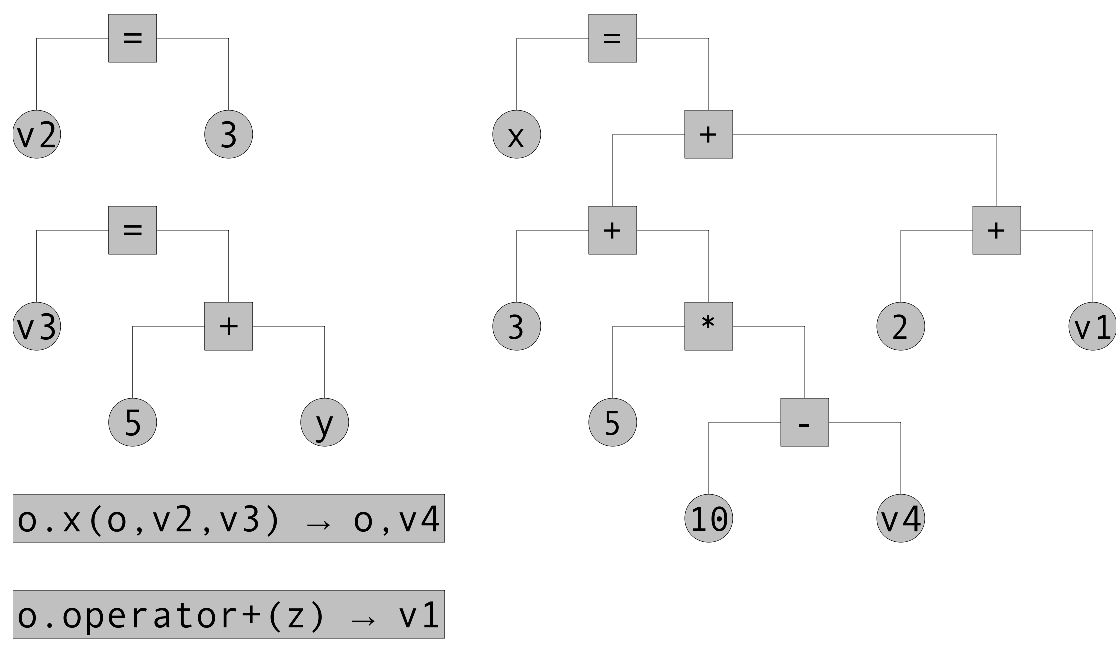


Illustration 5: Multiple trees from the single expression

These expression trees are then translated into the following xIL code (for simplicity reasons instead of variable numbers, the variable names are used in this piece of code):

```
1 eval v2 = 3
2 eval v3 = 5 y +
3 call o.x(o,v2,v3):o,v4
4 call o.operator+(o,z):o,v1
5 eval x = 10 v4 - 5 * 3 + 2 v1 + +
```

Text 21: Translated expression in xIL

Note that the xIL expressions are in postorder notation to allow easier checking and further analysis.

5 Abstract Interpretation

Although the principle of abstract interpretation has been formalized and explained in [Cousot77] because this tool is not well known outside the domain of static analysis, it will be introduced in this chapter with respect to its use in the Crosscheck system.

Abstract interpretation formally defines a way to generally reason about certain aspects of the program using technique which can be informally described as partial (or approximative) execution.

This technique provides excellent formalism for a wide variety of static analysis which due to the nature of abstract interpretation partially overlaps with problems which using other means would be deductible only using dynamic analysis techniques. It's biggest drawback is that the abstract interpretation can be slow, memory demanding and for certain languages the construction of rules and transitions for the abstract interpreter can be very hard.

Therefore, based on [Cook08] the xIL has been designed with the abstract interpretation's needs in mind to allow relatively simple semantics and interpreter construction.

5.1 Basic principles

The basics of abstract interpretation will be demonstrated on a classical static analysis problem – the constant propagation[23] [Muller05]. At first we need to specify the domains for variables. These domains are subsets of all possible values the variable may hold selected with respect to the problem one wants to solve. These domains must be ordered to form a lattice, which for this particular problem is very simple and is shown in the following figure.

23 The problem is to decide which variables can be replaced with constants, which yields much faster and smaller code.

In the lattice, the capital T corresponds to any, or unknown value which means that the value may be anything from defined range of the type and {} represents no value which means that no value has yet been set to the variable. Particular numbers corresponds to constants with the given value.

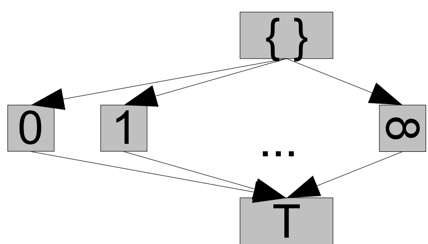


Illustration 6: Constant Propagation Domain Lattice

When the domain lattice is finished we need to redefine all operations and their outcomes to work with the already defined domains. Assuming the simple programming language has only three operators, = (assignment), + (addition) and – (subtraction), their tables are displayed below (*x* and *y* means any constant values):

=	{}	x	T
{}	!	x	T
y	!	x	T
T	!	x	T

+	{}	x	T
{}	!	!	T
y	!	x+y	T
T	!	T	T

-	{}	x	T
{}	!	!	!
y	!	x-y	T
T	!	T	T

Table 11: Operator tables for constant propagation

The operator tables show the output domains of the operations (left operand in rows, right in columns). Each cell defines the outcome domain and ! denotes invalid operation (the demonstration language is dynamically types as is xIL[24]). Apart from unapproved results it is clearly visible that assignment of constant and addition and subtraction of a constant are the only operations that result in constant outcome.

Additional operators including relational operator must be also specified in the same fashion. Clearly comparison of two constants can be correctly evaluated, comparison when any of the operands is in domain T will result in either *true*, or *false*, i.e. in T (we are treating boolean values as integers as does xIL).

When all domains and operators are defined we can proceed to construct the abstract interpreter which is very similar to classic interpreter. There are however some significant differences:

24 For languages like C/C++ replace ! with T's to obtain correct semantics.

1. All operations are performed in already defined abstract domains, not the concrete values.
2. Particular code block is executed repeatedly only if its last execution changed domain of any variable.
3. When two (or more) possible outcomes of a control flow are possible, both must be interpreted in parallel (this makes abstract interpretation quite demanding).
4. This leads to possible situations where a variable may be in two states at once. This nondeterminism is solved using the domain lattices.
5. A notion of reachability, i.e. if during the execution at one point a variable can have more values, their upper bound should be used instead.

To show the principle of the abstract interpreter, the following code will be examined according to the above defined rules:

```
1  i=5
2  j=10
3  k=i+j
4  while (i<j) {
5    k=k+i
6    i=i+1
7  }
```

Text 22: Constant propagation example

The process of abstract interpretation is illustrated in the following figure:

```
1  i=5                i=5,j={},k={}
2  j=10               i=5,j=10,k={}
3  k=i+j              i=5,j=10,k=15
4  while (i<j) {     1                1                       T
5    k=k+i            i=5,j=10,k=20   i=5,j=10,k=T (20,26)
6    i=i+1            i=6,j=10,k=20   i=T,j=10,k=T (5,6)
7  }
```

Text 23: Abstract Interpretation

The above example displays the first stage of abstract interpretation, domains (*after* the interpretation of current line) are shown in the right column. After the first iteration of the cycle the domains of *k* and *i* are changed to T (upper bound of first and second time pass). Therefore in the third iteration of the cycle the result of the condition can be anything (indicated by domain T) and thus the cycle must both be and not be taken. One branch will finish immediately and the other will pass the cycle body once more only to discover that no domains were changed and therefore terminates too.

The final result of (*i*=T,*j*=10,*k*=T) shows that only the variable *j* can be replaced with constant 10. The optimized code is displayed below:

```
1  i=5
3  k=i+10
4  while (i<10) {
5    k=k+i
6    i=i+1
7  }
```

Text 24: Sample program after constant propagation

Although the results shown in this example can be obtained by using much simpler techniques (for example counting writes to particular variables, variable with only one write of defined value can be replaced by constant), abstract interpretation allows greater precision (limited only by the number and organization of the domains theoretically converging to full interpretation). The last example in this chapter, while escaping the detection by the simpler method will still be updated using abstract interpretation, while the above presented method would consider both *k* and *j* as true variables.

```
1 function test(i) {         i=T,j={},k={}
2   j=10                     i=T,j=10,k={}
3   k=5                      i=T,j=10,k=5
4   if (i<10) {              0                1
5     k=j-5                                   i=T,j=10,k=5
6   }
7   j=k-j                    i=T,j=-5,k=5     i=T,j=-5,k=5
8 }
```

Text 25: More complicated constant propagation

The above example also shows another property of abstract interpretation – careful inspection of the results shows, that abstract interpretation actually simplified the code (all expressions modifying constants can be left out) during its single pass.

5.2 xIL Abstract Interpretation

The main disadvantage of abstract interpretation is that it may lead to very complex structures and superlinear complexity due to its nondeterministic nature. Therefore the algorithm of abstract interpretation used in Crosscheck has been slightly modified so that it still performs correctly in the context of plagiarism detection yet remains linearly complex.

This speed-up has been achieved by ignoring multiple jump instructions in xIL and introducing context-aware operator tables (i.e. operation may have different results depending on the interpreter context. The fact that multiple instructions are ignored means that at a particular line a jump instruction can be followed only once, thus limiting the passes of any cycle to 2. Additionally context sensitive operators allows less precise evaluation[25] of statements inside loops to compensate.

Also xIL interpretation treats inparallel blocks as if they are not really performed in parallel, but rather sequentially. This assumption is sound provided either of the two following rules are met:

25 Less precise in this context means with weaker supremal operator.

1. The inparallel blocks are mutually exclusive, which means that at the same level if one inparallel branch is taken the others are not. This means that if the branch contains any assert instructions, they will be evaluated to true,
2. or that the branches are data independent, e.g. if one branch reads certain variable, no other branch can write to it and no two branches can write to one variable.

These rules are actually a generalization of rules for safe parallel execution [Tvrdik00]. It is not surprising that xIL meets both of the criteria (inparallel blocks are either used to code mutually exclusive branches (if, elif, etc.), or to parallelize code (which can be done only if no data hazards are present). At the end of each inparallel block a supremum of each branch execution is computed for each variable.

The following example shows simple xIL code with a cycle and an if clause to demonstrate interpretation process. Additionally the above mentioned operators are updated in a way that when any operation is performed inside the loop which takes into account any uncertain variable, it's result is also uncertain (i.e. equal to T) and that any evaluation which reads and writes into the same variable also sets its abstract value to T:

```
 1 eval v1=1                 (v1=1,v2={})
 2 eval v2=0                 (v1=1,v2=0)
 3 sequence:                                  (v1=T,v2=0)
 4   inparallel:
 5     sequence:
 6       assert v1>2         False           maybe True
 7       eval v2=v1+2                        (v1=T,v2=T)
 8     sequence:
 9       assert v1<=v2       True            maybe True
10       eval v2=0           (v1=1,v2=0)     (v1=T,v2=0)
11   assert v1<10            True            maybe True
12   eval v1=v1+1            (v1=T,v2=0)     (v1=T,v2=T)
13   jump 3                  Taken           not taken
```

Text 26: xIL Abstract Interpretation Example

Clearly the classic abstract interpretation would have taken the jump at line 13 at least once more and it is not hard to imagine code that would take much more iterations to process correctly.

6 xIL Code Analysis

After the code is transformed from the source language into the xIL several static analysis tests are performed on the code in order to determine its most important and distinguishing features. While they are performed during only one abstract interpretation analysis in Crosscheck, they are explained separately for their better understanding. This chapter describes the most significant of them.

6.1 Code Importance

Code importance is a new concept developed for Crosscheck and is defined as a function that assigns to each line of code an integer representing the importance of that particular statement. The greater the number the greater the importance of the code line.

Although the code importance is computed only for the xIL program, code importance of lines in source programs can be easily calculated as sum of importances of all xIL code lines which were generated during the translation of the particular source code line. To remain fair proportions meta instructions are excluded from the summation as their amount per translated statement is arbitrary and depends on the statement's structure rather than its importance.

Code importance also does not have any upper bound, only the lower bound is defined to be 0. Such score indicates that the line can be removed from the code without changing the semantics of the program.

Code importance is computed during the abstract interpretation and does not require any domains and operator specifications. It uses the following rules to determine the final importance:

Rule	**Importance Calculation**
`eval v = ?` `exec (?):v`	These instructions writes value to the variable v. Every time a variable is being written the location of the instruction is remembered.

Rule	Importance Calculation

`eval ? = ?v?`
`exec(?v?):?`
`call v(?):?`
`call ?(?v?):?`
`assert v ? ?`
`assert ? ? v`
	Instruction which read from a variable will increase the importance of instruction which has set the read variable by the number of reads.

`eval v = ?v?`
`call v(?):v`
`call ?(?v?):v`
`exec (?v?):v`
	Any instruction that both reads and writes into the same variable will also remember the last instruction to write to the variable. Whenever its importance is increased, the importance of the original instruction is also increased.

`call fnc(?):v`
…
`return w`
	In the return and call instructions the address representing last change of variable w is remembered also for the variable v in the caller's context.

`call fnc(?):?`	Call of a determined sequence multiplies the importance of that sequence by two.
`call ?(?):?`	Call to unknown sequence (determined by non constant variable) multiplies the importance of any references sequences by two.
`jump forward`	Jump only
`jump backward`	Each jump (indicating taken cycle) multiplies all instructions from that jump target address to the jump itself by two. If there are any function calls or another backward jumps in the path, their rules are also applied.

Table 12: Code Importance Analysis Rules

6.2 Code Reachability

Generally speaking, any code with importance equal to zero is not reachable in a sense that such a code is not only unimportant but can be omitted completely without any loss of semantic precision. The following example shows only a few examples of unreachable statements in Python programming language with their brief explanations:

```
1 def fnc1(x): # this function is never called
2   print “I am function 1”
3   x=x+3 # moreover value from this expression is never used
4
5 z=4 # this expression's value is lost at the next line
6 z=6 # ...without being used previously
7 if (z<0): # and since z is a constant, this would never happen
8   print “z<0”
```

Text 27: Code Reachability Example

Additionally, when a function pointer is called, the pointer may either be fully specified in which case the respective sequence is called as would happen with any immediate sequence call, or multipliers of all instructions inside all referenced sequences are updated. This method may result in slightly higher importance values for pointered function. However, in classic programs such functions are usually callbacks, or dynamically selected functions both of which are usually very important itself.

As was already mentioned in previous chapters, any unreachable code automatically classifies the whole submission as plagiarized or at least highly suspicious.

6.3 Variable (not constant) Propagation

The third most important Crosscheck's analysis is the variable propagation. The test is called variable propagation since xIL does not have the notion of constants (immediates cannot be call or exec parameters, nor can they be returned from a sequence. They can only be evaluated to variables which in turn can be passed). This system was designed to filter any constants (either language specific or literal) into variables which can then be analyzed.

Each variable has a constant flag which means that the variable has been assigned value only once (and fits in the concrete domain). At the end of the analysis all constant variables are examined and their importance (importance of the constant variable is clearly the importance of the single line evaluating its value) is summed up according to their values. This results in an ordered list of constant values (not variables) and their importances, which is stored together with the submission for later analysis.

6.4 Program Flow Analysis

The last of the performed analyses is the program flow analysis. This analysis examines the program run (sequence of instructions in their execution order) and based on its statistical properties then creates its output string which is forwarded to the comparison engine.

The output string can be modified using various settings for better performance and/or efficiency. Some portions of this setup may be determined automatically by Crosscheck. This feature is discussed in greater detail in the next chapter.

In its most common form the output string is a linear representation of the program flow of the more important instructions. Additionally variable names and immediate values are all replaced with single letters as are the instructions.

6.5 Abstract Interpretation Specifications

While the general issues involved with abstract interpretation of xIL used in Crosscheck has been stated at the end of previous chapter, this chapter introduces their implementations and particular details.

6.5.1 Contexts

The code importance analysis distinguishes two contexts – normal and cycle context. Cycle context is active from the time a backward jump is taken till that exact jump (instruction address) is reached again. During the cycle mode different abstract domain lattices are used and also the rules for importance assessment are slightly changed.

Each instruction inside a loop is flagged so that the smallest possible importance increase in the future will be 2, not 1 (hence it's importance will be twice as high). The lattices used inside the loop are more general than the normal ones to compensate for the fact that each jump is taken only once during the abstract interpretation.

6.5.2 Abstract Domain Lattices

Although the abstract interpretation formally requires the variable to have a value from only one lattice, in Crosscheck a value is characterized by a vector of abstract values. Each operator is then defined for each of the vertices in the value vector.

The first part of the value vector is represented by the following lattice:

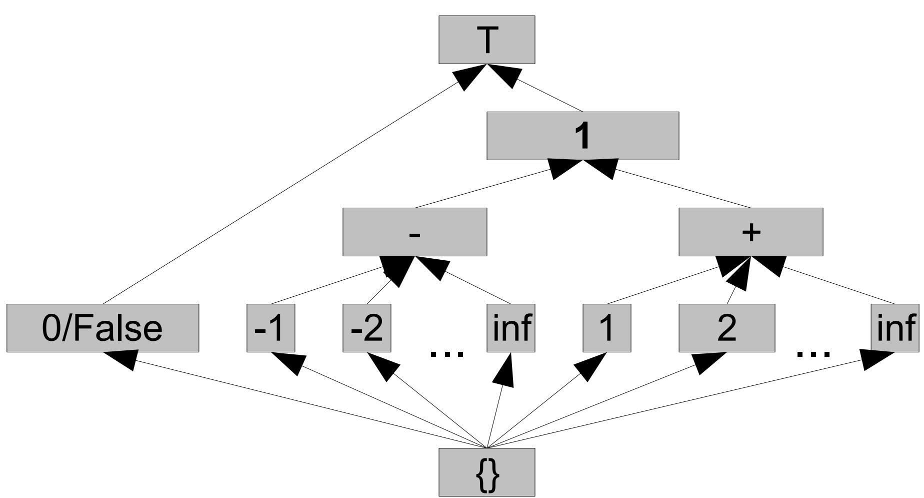


Illustration 7: Abstract Domains Lattice

The notion of the lattice is only an upgrade of the constant propagation lattice presented in the previous chapter. Not only are we interested in the actual value of the variable (to determine whether it should be included into the variable propagation analysis), but we also need to

know as precisely as possible the results of boolean expressions in assert instructions to determine the control flow.

In the following tables as well as in the lattice above 0 or False corresponds to the known value of 0. x+ corresponds to the known positive integer value, x- is the known negative value. + and – are unknown positive and negative values respectively. Bold **1** represents unknown value different from zero and T represents the universe, i.e. any integer value, positive, negative, or even a zero.

Additionally any string is represented as a bold **1** with the exception of an empty string which translates to 0.

Abstract operator transitions are defined in the following tables (only the interesting operators showing some concepts are illustrated).

Tables for relational operators are defined similarly exploiting the fact that positive number is always bigger than negative, etc.

+	{}	0	x+	+	x-	-	1	T
{}	!	!	!	!	!	!	!	!
0	!	0	x	+	x	-	1	T
y+	!	y	x+y	+	x+y	T	T	T
+	!	+	+	+	T	T	T	T
y-	!	y	x+y	T	x+y	-	T	T
-	!	-	T	T	-	T	T	T
1	!	1	T	T	T	T	T	T
T	!	T	T	T	T	T	T	T

-	{}	0	x+	+	x-	-	1	T
{}	!	!	!	!	!	!	!	!
0	!	0	x	+	x	-	1	T
y+	!	-y	x-y	T	x-y	-	T	T
+	!	-	T	T	-	-	T	T
y-	!	-y	x-y	+	x-y	T	T	T
-	!	+	+	+	T	T	T	T
1	!	1	T	T	T	T	T	T
T	!	T	T	T	T	T	T	T

*	{}	0	x+	+	x-	-	1	T
{}	!	!	!	!	!	!	!	!
0	!	0	0	0	0	0	0	0
y+	!	0	x*y	+	x*y	-	1	T
+	!	0	+	+	-	-	1	T
y-	!	0	x*y	-	x*y	+	1	T
-	!	0	-	-	+	+	1	T
1	!	0	1	1	1	1	1	T
T	!	0	T	T	T	T	T	T

%	{}	0	x+	+	x-	-	1	T
{}	!	!	!	!	!	!	!	!
0	!	!	!	!	!	!	!	!
y+	!	0	x%y	T	x%y	T	T	T
+	!	0	T	T	T	T	T	T
y-	!	0	x%y	T	x%y	T	T	T
-	!	0	T	T	T	T	T	T
1	!	0	T	T	T	T	T	T
T	!	0	T	T	T	T	T	T

&	{}	0	x+	+	x-	-	1	T
{}	!	!	!	!	!	!	!	!
0	!	{1}	0	0	0	0	0	0
y+	!	0	{1}	{1}	{1}	{1}	{1}	T
+	!	0	{1}	{1}	{1}	{1}	{1}	T
y-	!	0	{1}	{1}	{1}	{1}	{1}	T
-	!	0	{1}	{1}	{1}	{1}	{1}	T
1	!	0	{1}	{1}	{1}	{1}	{1}	T
T	!	0	T	T	T	T	T	T

|	{}	0	x+	+	x-	-	1	T
{}	!	!	!	!	!	!	!	!
0	!	0	{1}	{1}	{1}	{1}	{1}	T
y+	!	{1}	{1}	{1}	{1}	{1}	{1}	T
+	!	{1}	{1}	{1}	{1}	{1}	{1}	T
y-	!	{1}	{1}	{1}	{1}	{1}	{1}	T
-	!	{1}	{1}	{1}	{1}	{1}	{1}	T
1	!	{1}	{1}	{1}	{1}	{1}	{1}	T
T	!	T	T	T	T	T	T	T

Table 13: Operator tables

Another important part of the domain vector is the indicator of the purpose of the variable which is used to determine possible variables for the variable propagation analysis. The lattice for this abstract domain has states corresponding to the following values:

- Written – variable has been written, but not read yet. This state in final analysis indicates unimportant variable which may be left out.
- Read – This indicates that the variable has not only been set, but also at least one read operation has been performed. Therefore the variable has its meaning. At the end of the analysis this state means that the variable is a candidate for variable propagation depending on its importance.
- ReWritten state denotes variable which has been written and read and then written again, but with defined value not dependent on its previous value. While this would indicate that the variable might be considered for constant propagation (different constants in different regions) for Crosscheck's purposes this variable is treated as a normal one (usually these variables come from bad programming style, not plagiarism techniques)
- Updated variable is variable either depending on another variable whose value is unknown, or a variable that has been updated (i.e. rewritten with value depending on its previous value (the simplest example being the expression v=v+1). This state indicates normal variable which should not be considered for variable propagation.

And the illustration of that lattice is in the diagram below:

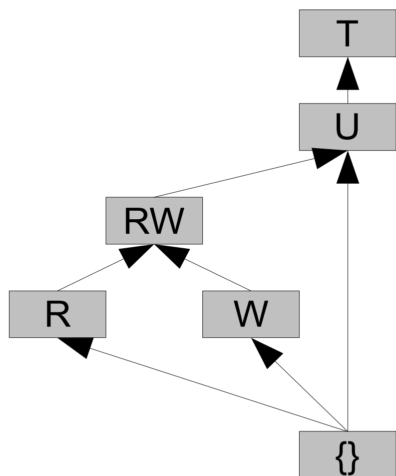


Illustration 8: Variable Propagation Domain Lattice

Obviously the only instructions capable of changing the value of this domain are evaluations, calls and execs (all of them writing to the particular variable). The following table shows the changes in the variable state based on its previous value and the type of instruction used:

State	`eval v=?`
	`call ?(?):v`
	`exec (?):v`
		`eval v=?v?`
		`call v(?):v`
		`call ?(?v?):v`
		`exec (?v?):v`
			`eval ?=?v?`
			`call v(?):?`
			`call ?(?v?):?`
			`exec (?v?):?`[26]

{}	W	!	!
W	RW	U	R
R	RW	U	R
RW	RW	U	RW
U	U	U	U
T	!	!	!

Table 14: Variable propagation rules

Now when all the formal parameters of the abstract interpretation have been specified, the following chapter demonstrates the abstract interpretation and analysis on slightly larger code.

26 And all other instructions reading from the variable v.

6.6 Analyses Example

For better understanding the Crosscheck's analyses are explained on language C, rather than its proprietary xIL language. Documentation on the analysis performed on xIL code in greater detain can be found in source code documentation.

Also note that the examples given in this chapter does not fully illustrate the potential of the analysis due to their size restraints. The more complex and structured the code is the greter impact the analysis has. However even mid-sized C programs are translated into xIL programs with hundreds of lines and their demonstration in this report is unfeasible[27].

6.6.1 Simple Example

For the purposes of this explanation, consider the following simple C code:

```
 1 void unnecessaryFunction() {
 2     fprintf("Unnecessary function");
 3 }
 4 void classicFunction() {
 5     fprintf("Classic function");
 6     int x=67;
 7     return x;
 8 }
 9 void functionInCycle(int j) {
10     return j-2;
11 }
12 int main() {
13     int z=classicFunction();
14     if (z<10) {
15         z=z+45;
16     } else {
17         printf("Always executed");
18     }
19     int y=z;
20     int c=0;
21     for (int i=0;i<10;i++) {
22         c=c+functionInCycle(i)+z;
23     }
24     return c;
25 }
```

Text 28: Analysis Example - C code

In this size, even to the naked eye some of its properties are easily spotted – for instance the fact that function unnecessaryFunction() is never used and can then be left out completely, or that the variable y at line 19 server no purpose at all. On the other hand, the observation that value of the variable x set at line 6 and then passed to z in the main function can be replaced with constant is not as trivial. Knowing this, it is easy to reason that the positive clause of if statement at line 15 will never

27 These more complex demonstrations can be found on the accompanying CD.

be executed (if it would be executed, z could not be replaced with a constant).

After the first phase, Crosscheck generates the xIL code, which is shown for future reference in the text below (for shorter code insignificant metainstructions have been removed):

```
 2    jump 37
 3    meta function "unnecessaryFunction"
 5    sequence:
 7        eval V2="Unnecessary function"
 8        meta name "fprintf"
 9        exec (V2)
11    meta function "classicFunction"
13    sequence:
15        eval V4="Classic function"
16        meta name "fprintf"
17        exec (V4)
22        eval V5=67
24        return V5
26    meta function "functionInCycle"
30    sequence(V7):
32        eval V8=V7 2 +
33        return V8
35    meta function "main"
37    sequence:
42        call 13():V10
43        eval V9=V10
45        eval V11=V9 10 <
47        inparallel:
48            sequence:
49                assert V11!=0
52                eval V9=V9 45 +
53            sequence:
54                assert V11==0
57                eval V13="Always executed"
58                meta name "printf"
59                exec (V13)
64        eval V14=V9
69        eval V15=0
76        eval V16=0
77        sequence:
79            eval V17=V16 10 <
80            assert V17!=0
83            call 30(V16):V18
84            eval V15=V15 V18 +  V9 +
86            eval V16=V16 1 +
87            jump 77
89        return V15
```

Text 29: Translated Analysis Example

Comparison of the translated code with the original reveals that the following mapping between xIL and C variables is used (in order of appearance):

C name	xIL name
x	V5
j	V7
z	V9
y	V14
c	V15
i	V16

Table 15: C to xIL Variable Names

Other variable indexes in xIL are used for auxiliary variables as xIL is unable to call or execute subroutines with immediate parameters.

6.6.2 Interpretation

When the code is translated it is interpreted. The interpreter produces two significant outcomes:

1. Sequence of instruction in the order of their execution (abstract interpretation does not consider meta instructions to the flow). Each of these instructions has also associated its importance. And obviously all these importances are nonzero positives.
2. Record of all used variables, their abstract domains and their importances.

The interpretation itself reveals some of the properties of the code, notably it discovers that xIL line 52 (corresponding to C line 15) will never be executed. While this example is fairly trivial, due to clever domain allocation the interpreter will be able to detect also more complicated examples where the control variable is not in specific domain, as shown in the following fragment:

```
1 def fnc(i):
2     if (i<0):
3         return 10
4     else:
5         return 0
6 i=fnc(13)
7 i=i+56
8 if (i<0):
9   print “unreachable code”
```

Text 30: More complicated dummy code

While the abstract interpretation does not recognize uncalled functions in itself, due to the fact that these functions will not be executed, their importance will remain 0 which would make them easily detectable by the upcoming analyses.

The abstract interpretation can also deal with function pointers. If the function pointer value is from a specific domain the appropriate function

is called as if the address would be immediate (this is likely to be the most prevalent case). However if the function pointers are from other domains (+,-,1,T) their value cannot be determined. In this case all functions that have ever been referenced in the given context are called in parallel, which decreases the importance metrics.

6.6.3 Variable Propagation Analysis

After the abstract interpretation has finished, the variable propagation analysis commences. It searches over all variables used and determines whether they are candidates for constants (state R), or never used variables (state W). If a variable is never used, it's initializing instruction is also removed (i.e. its importance is set to 0). Although many of the auxiliary variables are also candidates for constants (they were initialized and read only once after which they are never used again), their importance is very low (only one read) and are therefore omitted.

The following table represents the states, abstract values and write points of the C variables[28] (the write points are in C lines although in reality these parameters are in xIL terms). Important lines are described in the test as well:

```
 4 void classicFunction() {
 5     fprintf("Classic function");
 6     int x=67;                      x writePoint to 6
 7     return x;                      x importance+1
 8 }
 9 void functionInCycle(int j) {     j writePoint to 21
10     return j-2;                    j importance +1
11 }
12 int main() {
13     int z=classicFunction();       z writePoint to 6, importance +1
14     if (z<10) {                    z importance +1
15         z=z+45;
16     } else {
17         printf("Always executed");
18     }
19     int y=z;                       y writePoint to 19, z importance+1
20     int c=0;                       c writePoint to 20
21     for (int i=0;i<10;i++) {       i writePoint to 21, importance +4
22         c=c+functionInCycle(i)     c writePoint to 22, importance +2
                              +z;  z importance +2 (cycle)
23     }
24     return c;                      c importance +1
25 }
```

Text 31: Variable Analysis in Code

C name	Importance	State	Value	Write Point
x	6	R	67	6
j	2	U	+	21
z	4	R	67	6

28 The table for xIL would slightly differ due to auxiliary variables and redundant reads.

C name	Importance	State	Value	Write Point
y	0	W	0	19
c	3	U	+	22
i	4	U	+	21

Table 16: Variable Propagation Analysis Results

This analysis found that variable y is assigned but never used. Therefore the importance of line 19 in C (line 64 in xIL) has been set to 0.

Additionally variables x and z are identified as possible constants (their state is R and their value is from a specific domain). The analysis thus continues producing the list of possible constants ordered by importance. In this simple example the only constant is 67. However one can easily imagine more complex output, such as the one shown in the table below:

Value	Importance	Variables
45	75	V1,V10,V34,V56,V80
2.13	34	V4,V11,V91
128	12	V2
2	6	V45

Table 17: Imaginary Variable Analysis Results

The table lists important constant values that have been found ordered by their importance and accompanies by referencing registers. Therefore whenever a variable from the right column is met in the xIL code it can be replaced with immediate value in the left column.

6.6.4 Reachability Analysis

After variable propagation helps identifying remaining unreachable or dummy code, reachability analysis is called to find overall information about the submission. Reachability analysis computes the importance of source lines and determines larger important parts of the xIL flow code. The reachability analysis is used by the flow analyzer to produce optimized string.

These parts can be also used to hash th submissions into a database for their future use and effective search in later submissions of the same topic (only pieces of the important code will be searched assuming that submission that does not have the important parts is clearly an original work)[29].

29 This feature is not yet implemented and reachability analysis only provides support for future improvements.

Any unreachable code found is reported and implies that the submission is flagged as plagiary (even if later comparison would fail to identify the original work).

Crosscheck's visualization of the reachability analysis is displayed below (importance shown from white (unreachable) to black (most important code). While some lines are simply not visited translated to xIL and therefore cannot gain any importance (displayed in white), others are translated to xIL and yet are not important (displayed in red). The latter constitutes the unreachable code:

```
 1 void unnecessaryFunction() {
 2     fprintf("Unnecessary function");
 3 }
 4 void classicFunction() {
 5     fprintf("Classic function");
 6     int x=67;
 7     return x;
 8 }
 9 void functionInCycle(int j) {
10     return j-2;
11 }
12 int main() {
13     int z=classicFunction();
14     if (z<10) {
15         z=z+45;
16     } else {
17         printf("Always executed");
18     }
19     int y=z;
20     int c=0;
21     for (int i=0;i<10;i++) {
22         c=c+functionInCycle(i)+z;
23     }
24     return c;
25 }
```

Table 18: Reachability Analysis Output

Based on the algorithm for code importance and reachability it is not surprising that the most important code has been found on lines 21 and 22 that are part of a cycle, followed by the function functionInCycle() which is vital for the cycle's body and the function determining the value of the variable z (which has been revealed to be a constant).

For larger code chunks, the importance of source lines is then copied to all xIL lines translated from that particular line, because pure xIL importance tends to be defragmented over the core evaluations.

6.6.5 Program Flow Output

The final analysis is the program flow analysis and consequent output. The analysis performs basic statistical examination of the xIL code and its

importance determining minimal and maximal values, average, and various percentiles both for the code and for the constants.

After the analysis the program flow obtained from abstract interpretation is the tokenized into a single condensed string which is ready to undergo the final comparison part.

The following rules are used in the tokenization:

Input	Output	Comments
assert	Ar_operator_v	
bind	Br1_r2	
call	Caddr_args_results Caddr(args):results	 *optional*
eval	E Ev{operators} EEv{values} EEv{expression}	*optional consecutive*[30] *optional operators only* *optional values only* *optional whole expression (postfix, infix)*
exec	Xargs_results XE(args):results	 *optional*
jump	J	*only backward jumps*
return	Rargs R(args)	 *optional*
sequence	S	*optional*
inparallel	P	*optional*
variable	v x (constant) # (constant)	 *optional* *optional constant value*
immediate	x # (value)	 *optional real value*

Table 19: Output Flow Rules

And the following figure shows the full output of the presented example in 100th and 80th percentiles with constant replacement (beginnings of instructions are displayed in bold):

```
S{Cx:vS{EvxX(v)EvxR(v)}EvvEvvx<P{S{Av!=x}S{Av==xEvxX(v)}}EvxEvxS{Evvx<
Av!=xCx(v):vS(v){Evvx+R(v)}Evvv+v+Evvx+Jx}S{Evvx<Av!=xCx(v):vS(v){Evvx+
R(v)}Evvv+v+Evvx+Jx}R(v)}
```

Illustration 9: Full Program Flow Output at 100th percentile

```
{{EvxEvx}EvvEvvx<{{}{Evx}}EvxEvxS{Evvx<Av!=xCx(v):vS(v){Evvx+R(v)}Evvv+v+
Evvx+Jx}S{Evvx<Av!=xCx(v):vS(v){Evvx+R(v)}Evvv+v+Evvx+Jx}}
```

Illustration 10: Full Program Flow Output at 90th percentile

30 Consecutive evaluation instructions may be omitted if required.

7 Comparison

Comparison is the last stage of the plagiarism detection. While the previous phases analyzed only single submission and therefore did nout output any information directly relevant for the originality assessment (with the exception of reachability analysis), the final comparison only compares the submissions amongst themselves on one by one basis.

During the past I have experimented with various comparator algorithms (notably the Running-Karp-Rabin Greedy String Tiling [Wise93] which has been used in many other plagiarism detection tools, including the Jplag). Unfortunately these algorithms resulted in an extremely high number of false positives.

Therefore the current version of Crosscheck uses another algorithm which is a simple modification of the well known string folding algorithm that has been used in many domains (notably the stringology and bioinformatics[31]).

The updates of this algorithm ensure that Crosscheck will be able to detect also more complex code changes such as the statement reordering.

7.1 Basic Algorithm

The extended description of the algorithm can be found in [Kolar04]. In general the algorithm computes the best fold of string one (length m) on string two (length n) (and vice versa). This is obtained using dynamic programming and array *m*n*. This array is initially filled with 0 and rule for it's update is:

$$M_{x,y} = max(M_{x-1,y}, M_{x,y-1}, M_{x-1,y-1} + match(s1_x, s2_y))$$

Where *s1* and *s2* are the two strings, *M* is the matrix, *max()* is a standard maximum function and *match()* is the matching function defined below.

The advantage of the updated algorithm is that it allows folding of parallel strings (these strings differ from normal strings so that each position of the string is either a single character, or a sequence of parallel

31 String folding is used to determine RNA, mRNA or protein folding.

characters). Parallel strings are computed by the comparator from the given program flow and known program structure. The *match()* function is defined on these parallel strings in the following way (it is therefore only an extension to classic matching function with boolean result):

$$match(a, b) = |a \cap b|$$

Where multiple items of the same value are allowed in the intersection (intersection on aabac and dbaa is baa in no particular order). This algorithm can be further enhanced with the idea that when inside parallel parts of the string the score is increased for each parallel line separately so that (aab|bba) and (aba bab) will not match at all whereas in the simple version they are total match.

The output of this comparison is the *M* matrix with filled values, maximal match and computed lengths of the inputs *s1* and *s2* (their compared lengths are not equal to their nominal lengths as some characters are required to code the parallel structure).

7.1.1 Diagonal Analysis

Although the above presented algorithm is fairly accurate for classic submissions, it has one crucial disadvantage as it is not capable of matching misplaced statements apart from those parallelized before (if and switch statements in general). Consider for example an original calling two functions A and B in this order and the plagiary that calls them in reversed order (assuming this transition is acceptable, i.e. there are no dependencies between these functions). This situation can be graphically demonstrated by the folding of two strings, *aabbbbbbbcccccccccaaa* and *aacccccccccbbbbbbbaaa*. Note that while these strings (both have length 21) are composed of equal characters, their *c* and *b* portions are misplaced. Traditional folding results are shown in the figure below (the darker the color the higher fold):

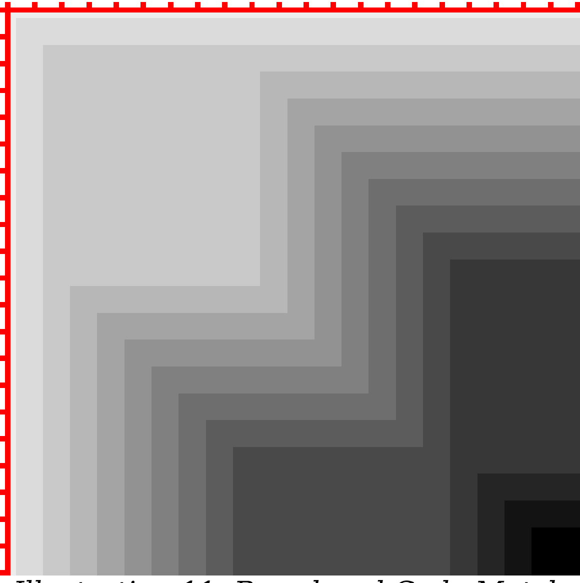


Illustration 11: Reordered Code Match

Because the final match is only 14 characters, which is less than 68% of the original length such a submission (in larger scale) would hardly be reported as plagiarized. To compensate for this situation, Crosscheck employs another upgrade to the algorithm, the diagonal analysis.

The matrix M is analyzed once more, to find the diagonals (i.e. lines along which the *s1* and *s2* match). Vertices on these diagonals are then replaced with the length of the diagonal (the number of consecutive matches) or with 0 none or only single match in a row occurs[32]. The output of this phase is shown in the following illustration:

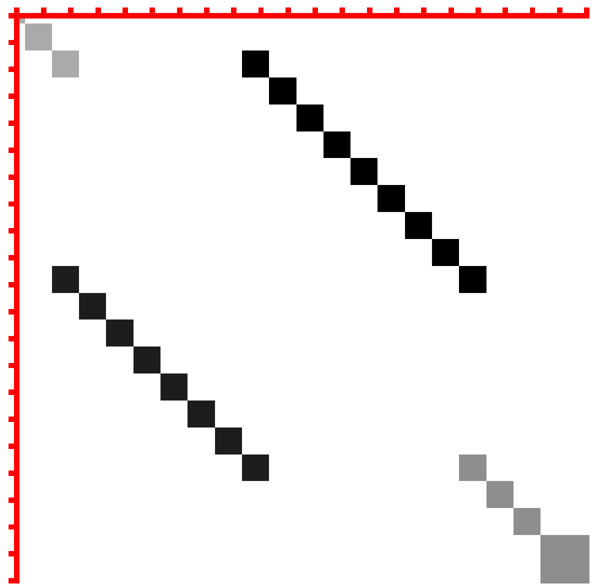


Illustration 12: Diagonal Analysis of Reordered Statements

The diagonals are then projected on the x and y axes in order to determine the tiling of the strings using the following rule:

$$T_i = max\{t / D_{i,j} \forall j\}$$

Where T_i is the tiling of *i*-th parallel character and $D_{i,j}$ is the matrix of diagonal analysis values. The final match is then calculated as number of parallel characters with are tiled (i.e. $T_i > \cdot$) and the likelihood of that match, which is average length of tile used (which is simply the average value of T_i). For the presented simple example the match is 100% which is excellent result.

The overall complexity of the algorithm is calculated below:

$$P = |s_1| \times |s_2| = m \times n = \Theta(n^2)$$
$$T = 2 \cdot \Theta(n^2) = \Theta(n^2)$$

32 Extremely small diagonals are not contributing to the result.

7.2 One to One Comparison Strategy

To determine plagiarized submissions Crosscheck must compare each submission with all others one by one. This is a lengthy process and increases the overall time complexity to $\Theta(n^4)$. This is the basic idea behind both human language and programming language plagiarism comparators.

However, the nature of programming language submissions and their cheating is rather different from the natural languages domain where this algorithm has been developed. While cheating in natural languages is usually done by paraphrasing other (sometimes not even topically close) sources (and paraphrasing many of them to produce the resulting essay), this technique is useless in computer languages due to the following facts:

1. Unlike essays, programming assignments are usually well defined with little or no space for creativity and originality.
2. Due to the much more restrictive syntax of programming languages, copying various algorithms from different sources requires great care to put them together in working order to produce the desired outcome.

Therefore most of the plagiarism in programming languages is done by applying modifications to only one source. This enables Crosscheck to slightly decrease the number of comparisons required. When a submission is found to be plagiarized, both the submission and its source are marked as plagiarized and none of them is checked against other submissions in the batch. Additionally these submissions are moved towards the beginning of a list from which new possible originals are drawn when checking the other submissions (as it is likely that one source has more then one copies).

The illustration below shows the difference between full and reduced search in a set of 30 short programs. Red squares represents plagiarized submissions (red square (x,y) means that submission x is plagiarized from submission y) and green shades reflect similarity of the submissions below the plagiarism threshold, which in this situation has been set to 90%:

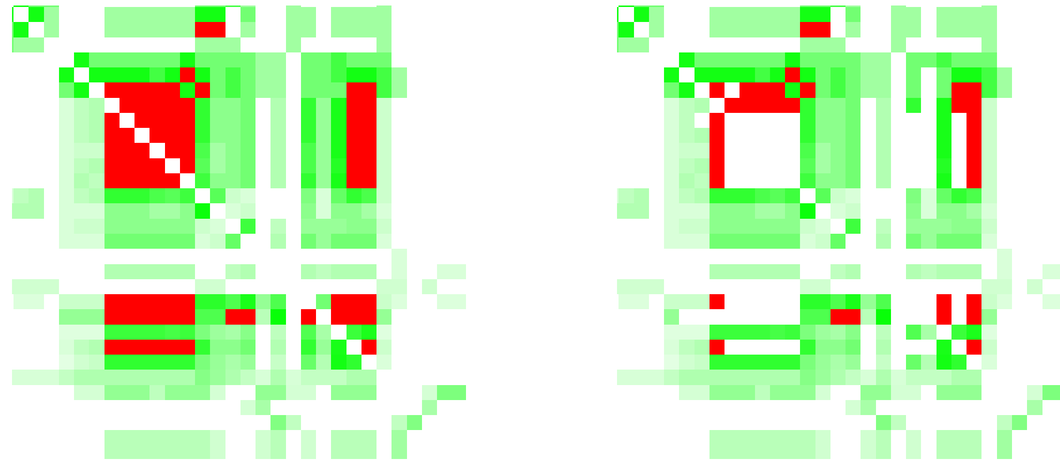


Illustration 13: Full and Reduced Comparison Visualization

Using this technique the number of comparisons required has been reduced to approximately 63% in the test set. Another possible enhancement is to assume that when source A has been suspected plagiary when compared with source B, then source B automatically is a plagiary too. While this assumption may seem sound at the first glance, one may easily imagine a very long and a very short submission with the short submission being composed only from parts of the larger submission. Clearly the large submission is the original while the smaller one is plagiary[33].

33 This can be safely assumed because Crosscheck's analyses eliminate redundant and unreachable code, therefore the all code in the larger submission has its meaning. The real example of such situation are two submissions, a QuickSort algorithm and a full data structure of a complex list capable of being sorted using QuickSort algorithm.

8 Evaluation and Results

This chapter attempts to evaluate Crosscheck's performance in two ways. At first the goals of the project are stated with following discussion regarding their completion. This is demonstrated on simple examples targeted only on the single feature. The second part of the chapter then deals with a supervised real world example in which a group of students was asked to submit the same coursework with half of them being instructed to cheat.

8.1 Crosscheck's Goals

Crosscheck's goals have been set as follows [Simecek08]:

1. The system must be able to handle variable and function renaming.
2. Capable of handling changes in function placement in the source code.
3. Crosscheck must be able (at least to some extent) identify dummy functions and variables (not used and not called ones)
4. The whole system must be easily extensible to allow processing of new source languages.

8.1.1 Variable and Function Renaming

Clearly Crosscheck is virtually immune to renaming of any kind because no names are preserved for the final comparison. However these can be always reconstructed from the xIL code to allow the reporter to pinpoint the changes.

Code altered only by means of renaming will be 100% compatible with the original.

This feature is also fully supported by all but the simplest alternative tools for plagiarism detection.

8.1.2 Function Placement Changes

Crosscheck is indeed immune even to these modifications as the program flow analysis groups the compared code together by means of the execution order, not the order they are defined in the source code. Hence the original and the plagiary will be again 100% compatible.

This feature is also supported by the alternative tools, however some more complex changes in the position may fool the comparators.

8.1.3 Dummy Functions and Variables

Even dummy functions and dummy variables can hardly fool Crosscheck. In most circumstances the program flow and variable propagation analyses either identify the unnecessary statements, or they are not visited at all due to the abstract interpretation of the source code. Programs trying to utilize this technique would most likely end 100% compatible in the final comparison.

This is the first feature possible due to Crosscheck's various analyses of the intermediate code and as such is not available in other tools for plagiarism detection.

8.1.4 Extensibility

Crosscheck's excellency in the above mentioned areas is possible partially as a trade-off in its extensibility. While adding a new language to other plagiarism detection tools (such as Jplag) mostly consists only of writing a relatively simple parser for that language, adding new language to Crosscheck is much more complicated.

To add a new language a new language parser and compiler to xIL must be developed. When the new language can be translated to the xIL all other Crosscheck's parts can remain the same. Additionally the complexity of the addition of a new language greatly depends on the language itself. While the extension for Java language would be one of the easier ones, addition of fully compatible C++ compiler to xIL is enormously complex task. On the other hand the existence of the intermediate language allows Crosscheck to possibly check for plagiarism even across supported programming languages.

Aware of the problems associated with its extensions, Crosscheck provides large SDK with many useful functions and template classes for easier development of parsers and compilers to xIL. These are documented in the source code documentation.

8.1.5 Additional Features

The powerful abstract interpretation in the heart of Crosscheck's algorithm gives it additional resistance against another and even more

advanced plagiarism techniques. In general while some of these features may be supported in other programs, Crosscheck's capabilities in this areas are orders of magnitude greater.

Variable Propagation

Crosscheck is to some extent able to detect variables that are used only as constants and replace them with their immediate values. These important constant values can be also ordered by their importance and added to the submission report. This allows to check not only whether source code has been plagiarized but also if the important values are not stolen, which may be desirable for tasks such as genetic algorithms where careful setup of the important probabilities is crucial to the algorithm's effectiveness.

Clever Dummy Code Insertion

Consider the following example:

```
1 int cleverFunction(int x) {
2   float z=0.5;
3   z=z**x;
4   if (z<0) {
5     // Large dummy code here
6   }
7   return z;
8 }
```

Text 32: Clever Dummy Code

Although it is not apparent, Crosscheck will correctly identify the dummy code at line 5 as unreachable and would not include it in the program flow output (although we know nothing about the variable x, we know that z is positive and positive number to the power of any other number is always positive, therefore cannot be smaller than zero). This code would be 100% similar to its original without the dummy code.

No other plagiarism detection tool is capable of such detection and most of them would be fooled by largely different submission size and its fragmentation due to such code pieces.

Statement Reordering

Statement reoredering (e.g. changing positions of statements inside functions) is one of the most advanced plagiarism techniques. Crosscheck is able to battle this technique on two fronts – using the xIL parallelization with inparallel instructions and diagonal analysis during the final comparison.

While string tiling algorithms in other tools can also correct the reordering, Crosscheck's algorithms work instruction wise which forbids tiles to span over instructions which are not covered totally.

8.2 Coursework analysis

The following observation was taken on high school students learning C/C++ for one year. Due to the fact that they are learning the language voluntarily to prepare themselves for higher education they can be roughly compared to freshmen university students.

8.2.1 Task Specification

A smaller task of finding way out of the maze has been selected to test Crosscheck's possibilities because smaller programs poses theoretically higher risks for the abstract interpretation as even small changes in the program flow would result in a significant percentage of altered code.

Precise definition of the problem which was also given to the students is displayed below:

> Create a program that will solve the maze problem, i.e. determine if there is a path from initial position to the gate of the labyrinth. The maze can be of any size and is represented by integers, walls are denoted by constant 100. The maze will always be bordered by walls, therefore you do not need to check constraints during the algorithm.

The following functions (implemented in maze.h and maze.cpp) were available to the students:

`int** maze_getLabyringth(int size)` which returns the generated maze problem.

`int maze_coordinateX()` and `int maze_coordinateY()` which returns the x and y coordinates of the actual position in the maze.

`bool maze_walk(int dx,int dy)` that performs the move in direction specified by dx and dy (only moves orthogonal to main axes are allowed and the person can move only one piece at a time). This function returns true if the move was successful and false if there is wall in the desired direction.

And finally `bool maze_finished()` returns true if the labyrinth gate has been reached, otherwise return false.

The students were instructed to either develop an original solution, or attempt to copy already existing solution in a way that would be undetectable by the automated detector. They were not told about Crosscheck's internal mechanisms, but they knew they would be facing an automated system, not a human.

Additionally I have added another submission to the repository. This contained insertion and selection sorts, i.e. a completely different program of roughly the same size.

8.2.2 Preliminary Analysis

After the submissions have been collected they were manually reviewed in order to identify their similarities which was possible due to their small amount and sizes. This analysis is summarized in the following table and

determines the base of the supervised analysis of Crosscheck's capabilities:

ID	Copies	Comments
1	6, 10	iterative modification 1
2		original work
3	8	recursive
4		sorts
5	9	iterative modification 2
6	1, 10	iterative modification 1
7		original work
8	3	recursive modification
9	5	iterative modification 2
10	1, 6	iterative modification 1

Table 20: Preliminary Analysis of the Submissions

Analysis of the submitted sources also revealed that most of the techniques very fairly simple and mostly limited to simple refactoring by the means of variable and function renamings. Some students altered, added, or stripped comments and from time to time more clever methods such as statement reordering have been used.

The most advanced form of plagiarism discovered was insertion of dummy code into active procedures (submission 6) and function inlining or extracting found in submission 10. Examples of both are provided below:

```
41          case 0:
42            if (canWalkLeft(maze,x,y)) {
43                maze_walk(-1,0);
44                maze[x][y]++;
45                break;
46            }
47            if (false) {
48               maze=0;
49               maze+=1;
50               printf("Posunuju se na dalsi krok.") ;
51            }
```

Text 33: Dummy Code Insertion into the tested submission

```
10 case 0:
11  if(maze[x-1][y]!=100) {
13       maze_walk(-1,0);
14       maze[x][y]++;
15       break;
16   }
```

```
6 bool canWalkLeft(int **maze,int x,int y){
7     if (maze[x-1][y]==100) {
8         return true;
9     } else {
10        return false;
11    }
12 }
47 case 0:
48   if (canWalkLeft(maze,x,y)) {
49     maze_walk(-1,0);
50     maze[x][y]++;
51     break;
52   }
```

Text 34: Inlining or Extracting Functions

Notably all three submissions are from the same source which is clearly visible even from their fragments already presented. However their detection by classical techniques is very unlikely due to the fact that each of these adds or removes relative large amounts of code (either used or unused one).

8.2.3 Crosscheck's Results

Based on my experiences with former versions of Crosscheck I have anticipated relatively large number of false positives and thus the similarity ratio at which the submission is considered to be plagiarized and should be reported has been set to 90%. All other settings has been left at their default values (this means minimal code importance of at least 2, constants replaced by their values, etc).

The best achieved results within this setup was either full, or partial symmetric algorithm (e.g. each two submissions are tested and the similarity between a and b is the maximum of their respective similarities).

At first the final comparison algorithm, the diagonal analysis was tested using its visualizations. Surprisingly the differences between match and no match are visible even to the naked eye, as shown in the following figure (mismatching pair at the top, matching below).

The are followed by the final table produced by the comparator. This table shows percentages of similarity for each pair of submissions. This value is also visualized using green shades for normal values and red shades for suspected plagiaries). A submission is reported as plagiarized if it has at least one red square either in a row, or in a column. Grey squares represent skipped comparisons, in this case (fully symmetric) it's only the main diagonal.

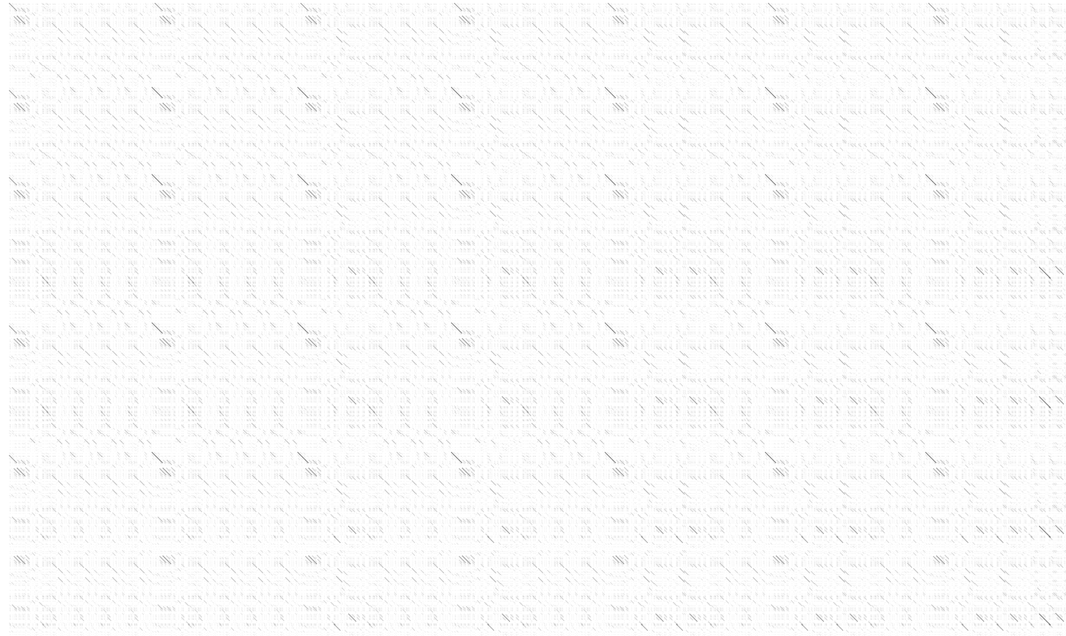
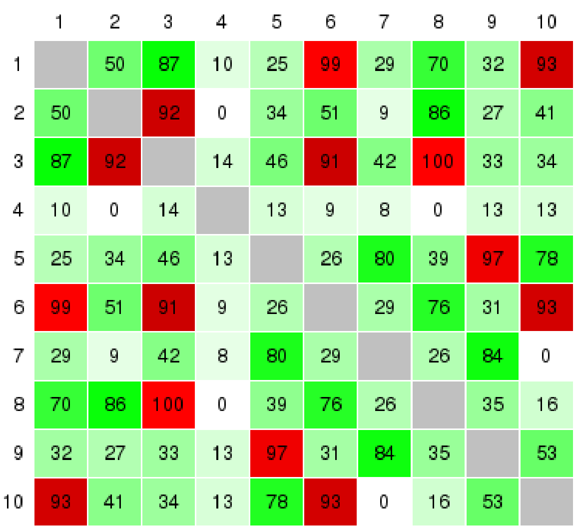


Illustration 14: Mismatching submissions (1,9)

Illustration 15: Matching submissions (1,6)



	1	2	3	4	5	6	7	8	9	10
1		50	87	10	25	99	29	70	32	93
2	50		92	0	34	51	9	86	27	41
3	87	92		14	46	91	42	100	33	34
4	10	0	14		13	9	8	0	13	13
5	25	34	46	13		26	80	39	97	78
6	99	51	91	9	26		29	76	31	93
7	29	9	42	8	80	29		26	84	0
8	70	86	100	0	39	76	26		35	16
9	32	27	33	13	97	31	84	35		53
10	93	41	34	13	78	93	0	16	53	

Illustration 16: Final Results

The final results show a remarkably well outcome of the analysis. The original submission #7 was indeed marked as original, three major clusters were also correctly marked (submissions {1,6,10} {3,8} {5,9}) and the submission #4 was clearly marked as unfit (notice the very low similarity ratios with other submissions). Due to the redundant code in submission #6 Crosscheck can even determine that this is likely *not* the original, a feature unique among other plagiarism detectors.

On the negative side, the Crosscheck seems to identified two false positives, namely the similarity between submissions 2 and 3 (92%) and between submissions 3 and 6 (91%).

8.2.4 False Positives and Their Explanation

To explain the found false positives, we will look at the asymmetrical version of the algorithm. This easily clarifies the second case because while submission 3 is very similar to the submission 6 (91%), submission 6 cannot be less similar to #3 than it is reported at 17%. Such varying results mean the only thing – submission 3 is much smaller than submission number 6 (remember 3 and 8 are recursive) and because it has many smaller parts similar with #6 is is indeed reported to be similar. On the other hand, because #6 is way too large to be a copy of submission number 3 and due to the fact that the matching sequences are very small in length compared to the overall length of the submission, its similarity result is almost opposite.

The second false positive is more interesting as the respective similarity is 60%. Although it is not as high as the other 92% it is high enough not to be dismissed by varying sizes. Closer look on the submissions' source codes reveals that both are in fact recursive and although they seem to be different, they share very similar portions, notably the recursion itself, as shown in the following text (submission 2 on left):

```
33 return (walk(-1,0) || walk(0,-1)
     || walk(1,0) || walk(0,1));
```

```
 8 if(solve(maze,1,0)) return true;
 9 if(solve(maze,-1,0)) return true;
10 if(solve(maze,0,-1)) return true;
11 if(solve(maze,0,1)) return true;
```

Text 35: Recursive false positive

8.2.5 Possible Improvements

Although this situation is theoretically one of the hardest tasks due to the extremely small programs and very precisely specified task and is thus unlikely to occur in large submissions (where the same effort was taken to cheat) it gives valuable hints for possible future improvements.

Notably the asymmetrical nature of the comparison should be exploited so that only one value (that equally reflects both similarities) will be issued for the pair. However to determine the exact way to achieve this on unsupervised data would certainly require much more extensive testing.

9 Conclusion

Crosscheck implements in many ways novel approach to the plagiarism detection in computer science programming courseworks. And while it has definitely proven that its path is worth exploring, as every new technology, Crosscheck is not error-free. This chapter attempts to assess Crosscheck's current state and its possible future development.

9.1 Future Developlment

In the foreseeable future, Crosscheck might additionally benefit from the following upgrades:

- Crosscheck would definitely benefit from more extensive testing on larger data sets (preferably of supervised data) and improvements to its xIL translator to fully meet standards of at least C/C++ and Java languages.
- better reporters (classes visualizing the analysis results) to cope with leading commercial products, such as turnitin.com
- while Python has proven to be excellent choice for the system itself and allowed fast development and prototyping, its drawback is lack of speed. Therefore reimplementation of bottlenecks (one by one comparison, abstract interpretation) into C/C++ might greatly speed up the whole application
- addition of other languages (Java, complete C++, Python, Pascal)
- integration into e-learning suite[34]
- as mentioned above also more extensive testing and evaluation which should render new improvements to the existing algorithms in order to soundly decrease the occurrence of false positives. Additionally Crosscheck should also be benchmarked against other plagiarism detectors so that it's accuracy can be compared.
- insertion of new stage that would utilize the important code segments to faster search of multiple submissions in a database to

34 This project is jointly developed by Tomas Nykodym and me.

determine most likely candidates for the one to one lengthy comparison.

9.2 Comments

Due to huge problems with xIL translators, the initially favored idea of unique intermediate language might be cost ineffective, or xIL can be disassembled from another language (Micrsosoft Intermediate Language in .NET, or assembler for gnu compiler). Although this change would require significant modifications of the current code and limit the set of possible languages to those supported by the used suite, only one translator to xIL would immediately support most of today's practical languages.

It might also be possible to change Crosscheck to be less accurate in the abstract interpretation as the loss of some of its most advanced functions seems of smaller practical value (such modifications would usually require fairly complicated cheating). This would in turn simplify the design of the translator and ease the Crosscheck's extensibility while retaining some of its advanced functions.

Personally, I believe that this is the course of actions that could improve Crosscheck and make it more competitive with other plagiarism detection tools while retaining most of its unique features.

References

[Aho06] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. **Compilers: Principles, Techniques, and Tools, Second Edition.** Pearson Education, Inc. 2006.

[Arwin06] Arwin, C., and Tahaghoghi, S. M. M. **Plagiarism detection across programming languages.** Tech. rep., School of Computer Science and Information Technology, RMIT University Melbourne, 2006.

[Burd02] Burd, E., and Bailey, J. **Evaluating Clone Detection Tools for Use during Preventative Maintenance.** Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation, 2002.

[Carbone01] Carbone, N. **Turnitin.com, a pedagogic placebo for plagiarism.** 2001

[Churchill05] Churchill, L. **Students: 2, turnitin: 0.** Mc Gill Daily, 2005.

[Cook08] Personal communication with Byron Cook during Second International School on Trends in Concurrency, Prague, June 22-27,2008.

[Cousot77] Cousot, P. and Cousot, R. **Abstract Interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints**, POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977.

[Dick02] Dick, M., Sheard, J., et al. **Addressing student cheating: definitions and solutions**, Annual Joint Conference Integrating technology into Computer Science Education, 2002.

[Eckel00] Eckel, B. **Thinking in C++**, Prentice Hall, Inc. 2000.

[Faidhi87] Faidhi, J., and Robinson, S. **An empirical approach for detecting**

program similarity and plagiarism within a university programming environment, In Comput. Educ. (1987), vol. 11, pp. 11-19

[Finkel96] Finkel, R., A. **Advanced Programming Language Design**, Addison-Wesley Publishing Company, 1996.

[Foster02] Foster, A. L. **Plagiarism-detection tool creates legal quandary**, The chronicle of Higher Education, May 2002.

[Grimes04] Grimes, P., W. **Dishonesty in Academics and Business: A Cross-cultural evaluation of Student Attitudes**, Journal of Business Ethics, February 2004.

[iParadigms07] **Plagiarism – technology**, iParadigms website, www.iparadigms.com

[Jones01] Jones, E.L. **Metrics based plagiarism motitoring**, In proceedings of the sixth Annual CCSC Northeastern Conference (2001)

[Kamiya00] Kamiya, T., Kusumoto, S., and Inoue, K. **A Token-based Code Clone Detection Tool - CCFinder and Its Empirical Evaluation**. Tech. Rep. Graduate school of Eng. Sci., Osaka University, Japan, 2000.

[Kamiya02] Kamiya, T., Kusumoto, S., and Inoue, K. **CCFinder: a multilinguistic token-based code clone detection system for large scale source code**. Graduate school of Eng. Sci., Osaka University, IEEE Transactions on Software Engineering, volume 28, 2002.

[Kolar04] Kolar, J. **Teoretická Informatika (Theoretical Informatics)**. Department of Computers, Czech Technical University, 2004.

[Lattner00] Lattner, C. A. LLVM: **An infrastructure for multi-stage optimization**. Master thesis, Graduate College of the University of Illinois, 2000.

[Lattner04] Lattner, C. A. and Adve, V. **LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation**, Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, Palo Alto, California, 2004.

[Maj08] Maj, P. **Originality Check Problem – Background Research**, Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University, 2008.

[Minsky67] Minsky, M. **Computation: Finite and Infinite Machines** (1st edition), 1967.

[Muller05] Muller, H. **Abstract Interpretation, Lecture notes for COMS30122 Advanced Language Engineering**, University of Bristol, United Kingdom, 2005.

[Prechelt00] Prechelt, L., Malpohl, G., and Philippsen, M. Jplag: **Finding plagiarisms among a set of programs**, Tech. rep. Department of Informatics, University of Karlsruhe, Germany, 2000.

[Schleimer03] Schleimer, S., Wilkerson, D. S., and Aiken, **A. Winnowing: Local algorithms for document fingerprinting**. Tech. rep., University of Illinois, Chicago, UC Berkeley, 2003.

[Schwartzbach03] Schwartzbach, M. I. **Lecture Notes on Static Analysis**, BRICS, Department of Computer Science, Univeristy of Aarhus, Denmark, 2003.

[Sheard02] Sheard, J., Dick, M., et al: **Cheating and plagiarism: perceptions and practices of first year IT students**, ACM SIGCSE bulletin, September 2002.

[Simecek08] Personal communication with Ivan Simecek. 2007-2009.

[Tvrdik00] Tvrdík, P. **Paralelní systémy a algoritmy**. Department of Computers,Czech Technical University, 2000.

[Turnitin07] Turnitin.com website, www.turnitin.com

[Wise92] Wise, M. J. **Detection of similarities in student programs: Yap'ing may be preferable to plague'ing**. Tech. rep., Department of Computer Science, University of Sydney, Australia, 1992

[Wise93] Wise, M. J. **String similarity via greedy string tiling and running Karp-Rabin matching**. Tech. rep., Department of Computer Science, University of Sydney, Australia, 1993

[Wise96] Wise, M. J. Yap3: **Improved detection of similarities in computer programs and other texts**. Tech. Rep. Department of Computer Science, University of Sydney, Australia, 1996

[Zhenmin05] Zhenmin, L., Shan, L., Suvda, M., and Yuanyuan, Z. **CP-Miner: A Tool for Finding Copy-Paste and Related Bugs in Operating System Code**, Department of Computer Science, University of Illinois, Urbana, 2005

Appendix A
Attached CD

Illustration 17: CD Contents

The above illustration represents the directory structure of the attached CD. Entire source code, reports and other related media are stored in the folder *Crosscheck*, while a backup copy is stored in *Backup*. All paths from now are relative to the */Crosscheck* directory:

Crosscheck's source code is stored in the folder */crosscheck* and its subfolders for each stage - */crosscheck/il* for intermediate language parsers and translators, */crosscheck/reports* for reporters,

/crosscheck/comparison for the comparators and */crosscheck/analysis* for the various analyzers.

Full source code documentation in HTML format generated by the Doxygen is located in */doxygen/html*.

/evaluation contains all data relevant to the Crosscheck's evaluation presented in this thesis. */evaluation/sources* contains the source codes of the checked submissions and other subfolders contain full analysis reports of the particular setup.

Finally the */reports* folder contains this thesis (in */reports/thesis*) and Crosscheck's presentation (in */reports/presentation*) in OpenOffice, Adobe PDF and postscript formats.

Appendix B Crosscheck's Brief Tutorial

Although Crosscheck is mainly intended as a Python library for the web based e-learning framework, a simple command line interface has been developed for the purposes of its evaluation.

System Requirements

Because Crosscheck is written entirely in Python programming language, it's requirements are only a few:

- Python SDK, should be compatible with any 2.x distribution
- Python Imaging Library for graphic outputs

Crosscheck has been tested with the following configuration:

- OpenSuSE 11.1 (x86_64)
- Python 2.6 (gcc 4.3.2)
- Python Imaging Library 1.1.6

Command Line Parameters

Crosscheck command line parameters are very simple. First parameter must be the prefix for output files. All remaining parameters are locations of the submissions to be checked. At least two files must be specified for successful start.

The following command[35]:

```
user@machine > crosscheck.py testPrefix s1.c s2.c s3.c
```

Evaluates the files s1 through s3.c and produces the following files:

- testPrefix_1_to_2.png, testPrefix_1_to_3.png, testPrefix_2_to_3.png which contains the graphical depiction of the diagonal analysis
- testPrefix_table.html that contains the final comparison table.

Crosscheck also contains numerous additional features such as advanced logging and configuration system. Documentation to these parts can be found on the accompanying CD in source code documentation.

File `demo.py` contains already configured script that will produce the results of the analysis performed in chapter 8.2 of this document.

35 use python crosscheck.py on Microsoft Windows machines.