JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Using feedback to improve black box fuzz testing of SAT solvers

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

INFORMATIK

Eingereicht von:
Jürgen Holzleitner, 0056645

Angefertigt am:
Institut für Formale Modelle und Verifikation

Betreuung:
Univ.-Prof. Dr. Armin Biere

Atzbach, Dezember 2009

Abstract

"Given enough time, a hypothetical chimpanzee typing at random would, as part of its output, almost surely produce one of Shakespeare's plays" (Wikipedia, 2009). This statement seems to explain why fuzz testing has become so successful since it was made public by Miller in 1990 (Miller, Fredrikson, & Bryan, 1990) and 1995 (Miller, et al., 1995). But as none of us can effort enough bananas to feed the chimpanzee until it finished typing, techniques have to be implemented to accelerate that task. This master thesis examines, based on applications in the field of satisfiability solving, whether black box fuzz testing can be improved by using feedback and adding methods common in the field of evolutionary algorithms. Intel's pin tool is used to get feedback about program behavior and this feedback is then used to construct further input by mutating existing ones.

Abstract

"Given enough time, a hypothetical chimpanzee typing at random would, as part of its output, almost surely produce one of Shakespeare's plays" (Wikipedia, 2009). Diese Aussage erklärt warum Fuzz Testing sehr erfolgreich eingesetzt wurde seit dem es von Miller 1990 (Miller, Fredrikson, & Bryan, 1990) und 1995 (Miller, et al., 1995) erstmals nachweislich verwendet wurde. Da es sehr kostspielig wäre, einen hypothetischen Schimpansen so lange mit Bananen zu versorgen, bis tatsächlich ein derartiges Stück als Teil des getippten produziert wird, müssen Techniken entwickelt werden um diesen Vorgang zu beschleunigen. Diese Masterarbeit untersucht ob black box fuzz testing von SAT solvern verbessert werden kann, indem Rückkopplungen und Methoden des Gebietes der Evolutionären Algorithmen angewandt werden. Dazu wird Intels pin tool verwendet, um Informationen über das Laufverhalten einer Testanwendung zu erhalten, und weitere Testeingaben werden generiert, indem vorhandene Eingaben, basierend auf den gewonnen Informationen, angepasst werden.

Keywords

Fuzz testing; satisfiability solving; black box testing; code coverage maximization; generational-based fuzzing; mutation-based fuzzing

Contents

1 Introduction 5

1.1 Fuzz testing (fuzzing) 5

1.2 Motivation 6

1.3 Maximizing code coverage as a goal 6

1.4 Subjects under test 6

1.5 The fuzzing life cycle 6

1.6 The baseline study 7

2 Developing the fuzzing application 10

2.1 Creating random formulas 10

2.2 Getting feedback from application runs 15

2.3 Description of the Monitoring tool 18

2.4 The fuzz application 22

2.5 The application main loop 22

2.6 Managing the state of the fuzzing progress 25

2.7 Mutation of already performed tests 29

2.8 Using feedback from test runs 34

2.9 Choosing the input to be mutated 38

2.10 Effect of checking for duplicates 53

2.11 Checking the effects of using an input cache 53

2.12 Conclusions of testing the benefit of the input cache 53

2.13 Resolving the memory problem 54

2.14 Final adaption of parameters 55

2.15 Final results of the tests with valid inputs 56

2.16 Comparison of the current strategy with cnfuzz 56

2.17 Further increasing the coverage 59

2.18 Final tests 67

2.19 Generation of the test suite 68

3 Final thoughts 70

3.1 Found Bugs 70
3.2 Given enough time 71
3.3 Porting the fuzzer to other applications 73
4 Conclusion 73
5 Bibliography 76
6 Appendix 78
6.1 Configuring the fuzz application 78
6.2 Installation on Linux based systems 79
6.3 Detailed description and code excerpts of the monitor tool 80
6.4 Killing an application 82
6.5 Creating non-uniform distributed random numbers 82
6.6 Monitoring performance analysis 83
6.7 Bash script to run cnfuzz 86
6.8 Description of the test environment 87
6.9 Details about the generated test suite script file 87
6.10 Analysis of cnfuzz 88
7 Lebenslauf 91
8 Eidesstattliche Erklärung 92

1 Introduction

"Is it possible to better find issues in software, using black box fuzz testing, if the amount of code covered during tests is maximized and what are suitable methods to increase coverage?" This is the main question that will be examined throughout this thesis by implementing and testing different strategies to fuzz SAT solvers.

Chapter 1 gives some basic information about the field of fuzz testing and code coverage. Additionally the subjects under test, which are SAT solving applications, are described and some information about a baseline study, used to compare results, are given. Chapter 2 describes a fuzzing application which was developed in order to tests certain strategies explored in this thesis. The chapter starts with the issue of generating random formulas which act as input to SAT solvers and are used as basis for further inputs. To get information about the code covered a special way of monitoring is used which is described next in this chapter. After these two basic techniques were introduced information about the structure of the developed fuzzing application are given whereas the focus is on creating new inputs considering previously generated ones. To perform this task methods, common in the field of evolutionary algorithms, are used whereas inputs which contribute to the amount of coverage are preferably selected and mutated in order to create new inputs. Throughout these sections many tests are carried out to check whether certain techniques and parameter sets are suitable. The chapter closes with the description of a test suite which is generated throughout the fuzzing process and contains all inputs which contributed to the overall coverage or yielded a problem in the tested SAT solver. Chapter 3 highlights the results of the work and describes issues which were found in the tested SAT solvers. Additionally some concepts to overcome the main limitation of the developed fuzzer, which is the required amount of time to process inputs, are described. Finally Chapter 4 summarizes the main findings of this thesis.

1.1 Fuzz testing (fuzzing)

According to (Takanen, Demott, & Miller, 2008) fuzz testing (fuzzing) is an instance of negative testing software. B. P. Miller is said to have invented fuzzing when he tried to find vulnerabilities in UNIX tools back in 1990 (Miller, Fredrikson, & Bryan, 1990). He did this by feeding the tools with random input and succeeded in breaking most of them. Since then fuzz testing has become an important technique to find software vulnerabilities. Many different kind of fuzz testing software have been developed: black box, gray box and white box fuzzing tools for a great number of different software and protocols. They all share the common characteristic to test a certain piece of software with a large number of random and more or less valid inputs. Fuzzing has been used successfully many times to find vulnerabilities in software as documented e.g. by (Granneman, 2006).

1.2 Motivation

"Intelligent fuzzing usually gives more results" (van Sprudel, 2005). Based on this statement this diploma thesis examines whether pure black box fuzz testing can be improved by some kind of feedback loop combined with evolutionary methods. The target applications are satisfiability (SAT) solving applications. Besides finding bugs, the developed fuzzing tool is able to generate a test suite with the goal of maximum code coverage. This way a software engineer is able to test many parts of his application in an automated way. The fuzzing tool does this by using nothing else than the binary of an application which makes it possible to check an application which is highly optimized and has all additional information, e.g. debug symbols or coverage information, stripped which is usually the case with SAT solving applications.

1.3 Maximizing code coverage as a goal

The primary goal of a fuzzer is to find bugs in software. In the case of fuzzing applications it seems obvious that flaws will be identified only if those parts of an application which cause the flaw are executed. Although higher code coverage does not necessarily mean that more bugs are found there may be a high tendency towards this assumption as documented in (Zeller, 2006). Based on this observation this diploma thesis tries to identify code flaws by finding ways to increase the level of code coverage while fuzzing an application.

1.4 Subjects under test

As already mentioned, the main subjects under test are SAT solving applications. To be more concrete the SAT solvers picosat and precosat (Biere, 2009), developed by Univ.-Prof. Dr. Armin Biere at Johannes Kepler University, will be used and tested throughout this thesis. The solvers are capable of taking input in DIMACS format (DIMACS Challange, 1993) and decide whether the specified SAT problem in conjunctive normal form (CNF) is satisfiable. As the source code of the solvers is available it is possible to deeper analyze certain behavior, such as line coverage, on a higher level during the development phase. Nevertheless the fuzzing application developed in this thesis is available to perform all tasks without using this additional information which makes it possible to test highly optimized versions of the SAT solving applications too. To get information about the amount of covered code in such applications a special kind of monitoring is implemented which traces the basic blocks that are executed at binary level.

1.5 The fuzzing life cycle

The fuzzing life cycle, introduced by (Takanen, Demott, & Miller, 2008) and depicted in Figure 1, shows the major steps to create a fuzzer. The interface to the application is more or less defined when SAT solvers are fuzzed although some additional issues, such as fuzzing arguments, need to be considered which is described in this thesis in Sec. 2.17. Input generation and sending inputs are described in Sec. 2.5.2 and Sec. 2.5.3. Target monitoring is

another crucial method when fuzzing applications which becomes very important in this thesis as it is used to get feedback from the application. It will be described in Sec. 2.2. The overall goal of fuzzing is to find bugs and therefore exception analysis needs to be handled which is also described in this section. Finally, reporting will be implemented as a way of generating test scripts that cover all the touched code parts which is described in Sec. 2.19.

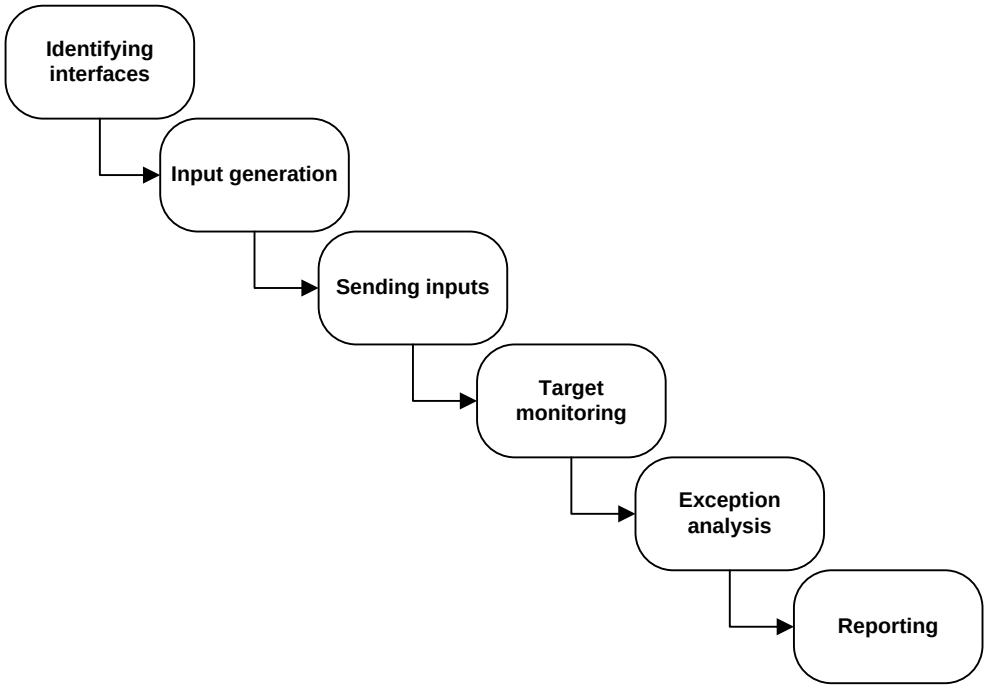

Figure 1: The fuzzing lifecycle. Taken from (Takanen, Demott, & Miller, 2008)

1.6 The baseline study

Univ.-Prof. Dr. Armin Biere, who developed the SAT solvers under test in this thesis, provided a fuzzing tool called cnfuzz which is capable of creating random SAT problems in CNF. The tool mainly defines a random model of a formula, as described in more detail in Sec. 6.10, which is then serialized in DIMACS format to standard output. This fuzzer is used as baseline value to check whether the techniques implemented in this thesis lead to useful results. To estimate the capabilities of cnfuzz an initial test is carried out that analyzes which parts of the source code were hit when the fuzzer is run for a certain amount of time. To get coverage information the gcov utility (GNU Free Software Fundation, 2008) from GNU's compiler collection is used. The utility requires that the binary is annotated with additional instructions, which is done by setting certain compiler flags. To be more precise GCC's compiler flags "–fprofile-arcs" and "–ftest-coverage" need to be added in the corresponding makefile prior to building the SAT solving application under test. To make test results comparable it is possible to specify a fixed number of test runs or a fixed amount of testing time. As there may be huge differences in processing times for different fuzzing strategies used throughout this thesis, the primary

attempt pursued to compare test results is to limit the amount of testing time. Sec. 6.7 describes a bash script which is used to run a specified application for a certain amount of time on Linux based operating systems. The script collects gcov information after every test run and appends it to corresponding files. For this test one hour was chosen as an appropriate amount of testing time for no particular reason. Within that time the cnfuzz tool was able to cover 67.966% of picosat's code lines with 43138 tests and 81.112% of precosat's code lines with 36412 tests. Figure 2 and Figure 3 show the change in code coverage over time. The logarithmically scaled number of tests is drawn on the x-axis and the overall percentage of covered code lines on the y-axis. A more detailed analysis of picosat's test run showed that the final coverage was reached after 21128 tests. The high amount of 67.462% had been reached already after carrying out only 1812 tests. The detailed analysis of precosat's test run showed that it took 11072 tests until the final coverage value was reached. The high value of 80.762% had been reached already after 3076 tests.

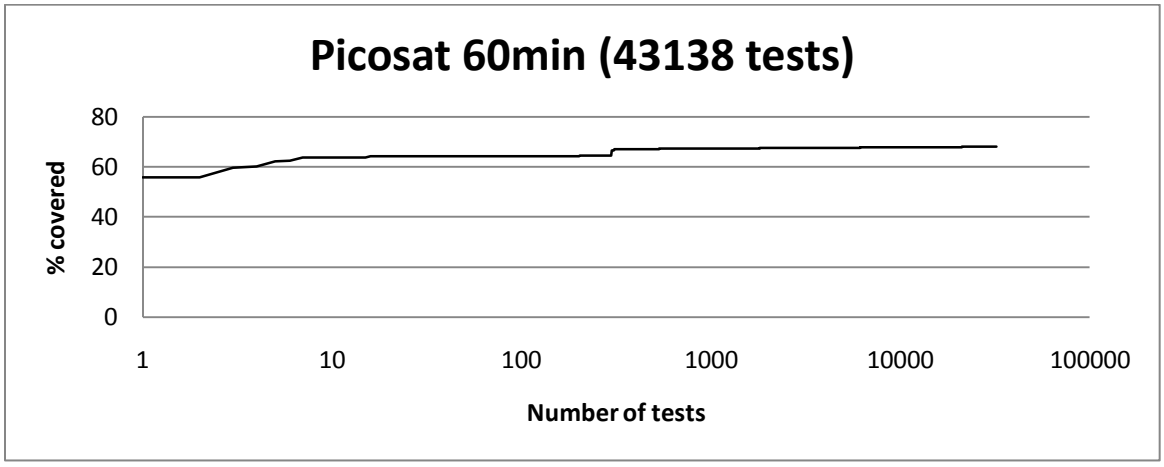

Figure 2: Testing picosat with cnfuzz for 60 minutes

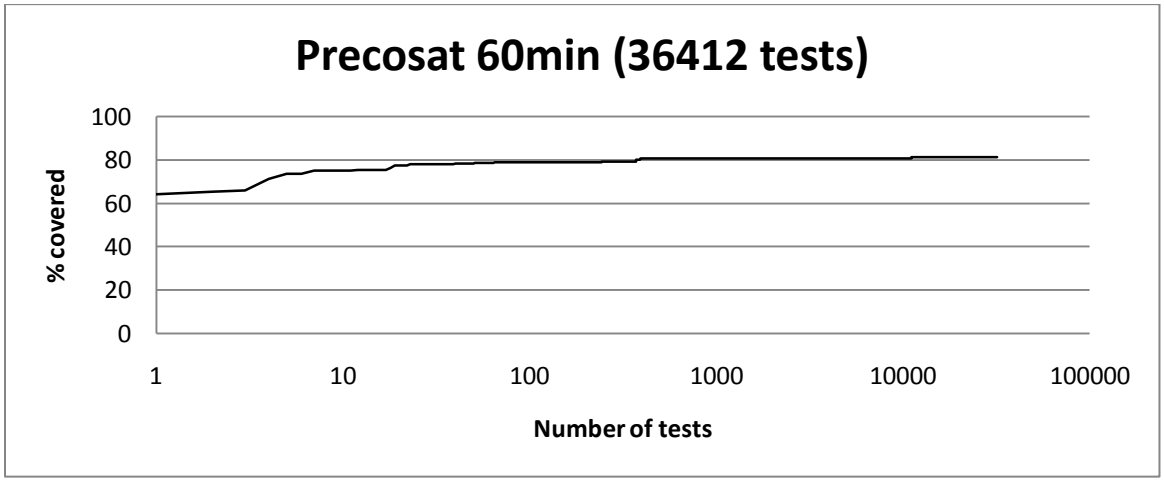

Figure 3: Testing precosat with cnfuzz for 60 minutes

1.6.1 Analysis of the code that was not hit

Further evaluation of the source code lines that were not covered during the tests just described yielded two main reasons for not executing many parts of the code. The first reason is that certain arguments are required to be passed to the application to enter certain code paths. The second reason is that some parts of the SAT solving applications are processed only if semi valid or totally invalid data is passed to the application. As the cnfuzz tool generates only valid input these parts were omitted from being executed. The creation of semi valid and invalid inputs as well as fuzzing arguments is described in more detail in Sec. 2.17. Besides the code that was not hit for the two reasons described, some parts in the code exist which were not covered because no suitable input was generated during the test. These parts will be targeted when feedback from application runs is used to generate test data of higher quality as described in Sec. 2.7.

2 Developing the fuzzing application

This chapter describes the strategies tested and or implemented in the developed fuzzing application. The explanations start with the generation of a model to create random formulas in CNF.

2.1 Creating random formulas

To make it possible to get and use feedback from application runs it is necessary to generate inputs to start with. As documented in (Takanen, Demott, & Miller, 2008) two widely used strategies are available which are to alter predefined inputs as used by mutation-based fuzzers or to randomly create new inputs as used by generational-based fuzzers. The fuzzer developed in this thesis combines both strategies and uses a generational-based approach to generate formulas and mutation-based approaches to further refine the most promising ones. This and the following sections are about the generational-based approach and describe a method to randomly create new formulas. The described method is based on the creation model used by the cnfuzz tool, which is described in more detail in Sec. 6.10. This tool was tested extensively in Sec. 1.6 and the results showed that the amount of code covered by the tool is already very high. Nevertheless a slightly refined and fully customizable model was created to test if further improvements are possible. The following sections make use of Gaussian as well as exponential distributed values. The techniques used to generate such values from uniformly distributed values, as they are provided by most random number generators, are given in Sec. 6.5.

2.1.1 Details of the random formula creation model

The creation of formulas is based on and starts with the number of different variables that will be used in the formula. This proved to be a suitable parameter to control the overall size of a satisfiability problem. The value is chosen from a Gaussian distribution as described by the following formula whereas n denotes the number of variables. m and δ_m specify the parameters of the Gaussian distribution.

$$n \sim N(m, \delta_m)$$

The number of clauses c in a formula is calculated by multiplying the number of different variables by a normal distributed factor with parameters f and δ_f. This is based on an analysis by (Mitchell, Selman, & Levesque, 1992) which resulted in the finding that satisfiability problems tend to be hard if the ratio between the number of clauses and the number of variables is within a certain range.

$$c \sim n * N(f, \delta_f)$$

The number of literals l in a clause is chosen exponentially distributed with parameter λ_l whereas a minimal length l_m is specified. Several tests showed that a minimum amount of variables in a clause is necessary to prevent the generation of formulas which could be proven

as unsatisfiable quickly. If longer clauses are generated by providing a smaller parameter to the exponential distribution a large amount of the clauses will tend to contain too many variables and they are proven as satisfiable very quickly.

$$l \sim l_m + \exp(\lambda_l)$$

In cnfuzz a model with layers is used to choose variables from neighboring areas and generate sets of clauses that connect them. To check if this improves the quality of the generated formulas a test was carried out where variables v of clauses are chosen equally distributed from the total number of available variables n. The sign of a variable is chosen equally distributed between true and false.

$$v \sim U(1, n)$$

The tests showed that modeling formula creation this way tends to create formulas which take more time to be solved by the tested SAT solvers compared to formulas created with the cnfuzz tool. But the analysis of code lines covered by the generated formulas showed that fewer code parts were actually executed by these formulas. This may be explained by the assumption that satisfiability solvers try to detect structures in formulas and make decisions on variables that connect these structures. That way it is possible to split the whole problem into smaller parts which can be evaluated faster. The use of uniquely distributed variables tends to create formulas which are highly unstructured and therefore the code which operates on structures in the formula is very unlikely to be executed. To create formulas which are more structured an initial variable m is chosen from all the available variables for each clause and the actual variables v in the clause are chosen normally distributed with parameters m and δ_v around this variable as depicted in Figure 4.

$$m \sim U(1, n)$$

$$v \sim N(m, \delta_v)$$

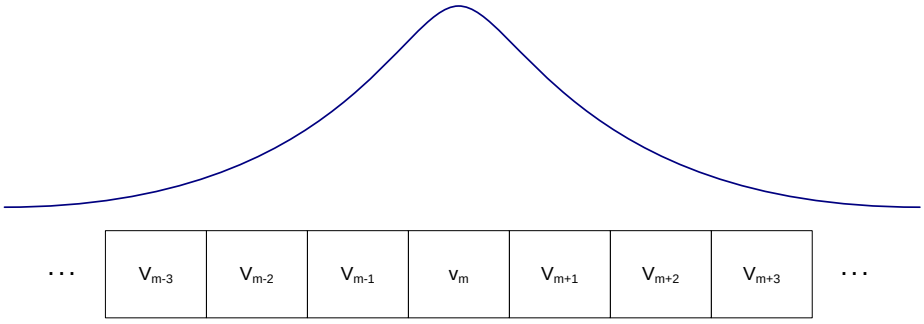

Figure 4: Selection of variables

If the selection of the variables is modeled this way it is also possible to approximate clauses with uniquely distributed variables by choosing a high value for the parameter δ_v and a wide range of different formulas can be created with this single model. Another question is whether prohibiting duplicate variables in clauses leads to useful results. Again, this is analyzed by carrying out some tests which resulted in the finding that no notable differences between the two strategies were encountered. Therefore it was not possible to decide experimentally if preventing duplicates in a clause is a valuable strategy. Due to the nature of formulas in CNF the underlying problem will most probably get easier if a variable occurs more often within a clause. Especially this is the case if a variable occurred in different phases within a clause which makes that clause satisfiable immediately. Due to this observation duplicate variables in clauses will be prohibited when formulas are created randomly. They may be created when mutators will change formulas which will be described in more detail in Sec. 2.7. Finally the creation model uses seven parameters and the creation process can be fully characterized by giving the parameters concrete values. Although it is not implemented in this thesis one may think of optimizing these parameters too while the application is fuzzed. This could be implemented by techniques common in the field of parameter optimization as described e.g. by (Bäck & Schwefel, 1993). The next section outlines a number of tests that were carried out in order to get suitable values for the parameters experimentally.

2.1.2 Getting suitable values for the random creation model

A number of different parameter sets were tested to get suitable values for the random creation model experimentally which are listed in Table 3 and Table 4. Each test was carried out for a time of 15 minutes and the used parameters as well as the corresponding line coverage reported by the gcov utility are depicted in these tables. The first two tables, Table 1 and Table 2, show the amount of covered code lines after using the cnfuzz tool for a time of 15 minutes and are used to compare results.

Test with cnfuzz for 15 min – picosat				
Tests				Coverage
Total	Sat	Unsat	Timeout	
9033	4924	4109	0	67.72

Table 1: Baseline test of cnfuzz - picosat

Test with cnfuzz for 15 min – precosat				
Tests				Coverage
Total	Sat	Unsat	Timeout	
7830	4154	3676	0	78.11

Table 2: Baseline test of cnfuzz - precosat

Tests with picosat for 15 min											
NumVariables		ClausesFactor		NumLiterals		Var	Tests				Coverage
Mean	Deviation	Mean	Deviation	Min	ExpMean	Deviation	Total	Sat	Unsat	Timeout	
1000	800	13.5	3.5	3	3	10	2620	1120	1500	0	66.73
100	**80**	13.5	3.5	3	3	10	11896	6135	5760	1	67.25
100	80	13.5	3.5	3	3	**50**	1424	774	550	100	67.67
100	80	**10**	**0.5**	3	3	50	16227	16164	63	0	66.87
100	80	**17**	**0.5**	3	3	50	1245	31	1116	98	67.50
100	80	13.5	**7**	3	3	50	2051	1040	915	96	67.72
100	80	13.5	**0.5**	3	3	50	778	460	203	115	67.67
100	80	13.5	7	3	**0.01**	50	20534	1619	18915	0	67.72
100	80	13.5	7	**2**	0.01	50	27237	568	26669	0	51.40
100	80	13.5	7	**1**	0.01	50	35181	277	34904	0	43.76
100	80	13.5	7	**5**	0.01	50	803	624	33	146	67.16
100	80	13.5	7	**8**	0.01	50	10295	10295	0	0	53.99
100	80	13.5	7	3	**5**	50	2142	1771	266	105	67.63
100	80	13.5	7	3	**8**	50	5719	5642	37	40	67.50
100	80	13.5	7	3	**12**	50	6797	6796	1	0	64.06
100	80	13.5	7	3	**1**	50	8157	1420	6701	36	67.72
100	80	13.5	7	3	**2**	50	3270	1021	2179	70	67.50
100	80	13.5	7	**2**	**4**	50	11956	2002	9954	0	63.55
100	80	13.5	7	**2**	**6**	50	9430	2745	6684	1	63.55
100	80	13.5	7	**2**	**8**	50	8293	3677	4616	0	63.55
100	80	13.5	10	3	1	50	8043	1195	6488	360	67.72
100	80	13.5	13	3	1	50	8512	1652	6839	21	67.50
100	80	13.5	**8.5**	3	1	50	6666	1255	5387	24	67.50
100	80	13.5	**5**	3	1	50	7551	872	6655	24	67.72
100	80	13.5	5	3	3	**10**	14194	1553	12641	0	66.73
100	80	13.5	5	3	3	**75**	6285	739	5529	17	67.67
100	80	13.5	5	3	1	**100**	6651	790	5841	20	67.76
100	80	13.5	5	3	3	**100**	1150	605	473	72	67.46
200	80	13.5	5	3	1	100	2066	196	1819	51	67.46
200	80	13.5	5	3	**3**	100	257	117	81	59	66.39
200	80	13.5	5	3	3	**50**	275	118	104	53	67.12
50	40	13.5	5	**3**	**4**	18	18298	14077	4221	0	67.72

Table 3: Random model creation - picosat

Tests with precosat for 15 min											
NumVariables		ClausesFactor		NumLiterals		Var	Tests				Coverage
Mean	Deviation	Mean	Deviation	Min	ExpMean	Deviation	Total	Sat	Unsat	Timeout	
100	80	13.5	5	3	1	100	4435	509	3899	27	80.948
100	80	13.5	5	3	**2**	100	1217	325	836	56	81.2426
100	80	13.5	5	3	**3**	**50**	932	486	377	69	81.3813
100	80	13.5	**10**	3	3	50	1295	608	624	63	80.3673
100	80	13.5	5	3	**4**	50	1056	808	180	68	80.8786
100	80	13.5	5	3	3	**25**	1736	938	752	46	81.7107
100	80	13.5	5	3	3	**10**	7389	3816	3573	0	80.9826
100	80	13.5	5	3	3	**18**	4526	2411	2110	5	81.7627
100	80	13.5	5	3	3	**30**	1307	694	548	65	80.9133
50	80	13.5	5	3	3	25	3495	1881	1579	35	81.3813
50	**40**	13.5	5	3	3	25	9328	5130	4196	2	81.78
25	**20**	13.5	5	3	3	25	25031	14405	10626	0	80.9826
50	40	13.5	5	3	3	18	10745	5871	4874	0	81.78
50	40	13.5	**3**	3	3	18	9781	5532	4249	0	81.676
50	40	13.5	**7**	3	3	18	10514	5529	4985	0	81.5287
50	40	13.5	5	**2**	**4**	18	23677	2979	20698	0	78.4171
50	40	13.5	5	**3**	**4**	18	9563	7363	2200	0	81.8147
50	40	13.5	5	**3**	**5**	18	10025	9179	846	0	81.676

Table 4: Random model creation - precosat

2.1.3 Analysis of the experiments

The tables, Table 3 and Table 4, show that good control over the created SAT problems is given. The mean amount of used variables gives control over the size of the created problems and if the number of clauses is set to create hard problems the processing time, required to solve the SAT problem, can be influenced. The fact that high line coverage and problems that require a relatively high amount of processing time are possible with as little as 50 different variables mean is a bit of a surprise. Based on the assumption by (Mitchell, Selman, & Levesque, 1992) it is no surprise that the tendency of the formula to be satisfiable or unsatisfiable can be controlled by the ratio between the number of clauses and the number of used variables. This work concluded that a ratio of 4.3 is a suitable value to generate formulas which are near the point where around 50% of the formulas are satisfiable and therefore expected to be hard to solve. In the paper all clauses had a fixed length of three variables. The fact that clauses in the generated code are longer by a certain factor, and thus easier to satisfy, explains why the optimal ratio between the number of generated clauses and the number of different variables in the tests are higher respectively. The tests also emerged that a minimum of three variables should be in a clause to have high code coverage. Fewer variables in a clause tend to make formulas proven as unsatisfiable very quickly without executing uncovered code. Removing variables to produce short clauses will be handled by mutation of existing formulas anyway as

described in more detail in Sec. 2.7. Having clauses that are significant longer than three variables, tends to make formulas proven as satisfiable quickly and the amount of covered code lines did not increase too. Another finding is that decreasing the local connectivity of variables chosen for a clause by increasing the value of the deviation when variables are chosen, tends to increase processing time necessary to solve the generated formulas but does not increase the amount of covered code lines.

2.1.4 Conclusion of the experiments

Compared to the cnfuzz tool the creation model yielded no significant gain in line coverage when picosat was fuzzed but increased the overall line coverage when fuzzing precosat by more than three percent which is a remarkable amount as depicted in Figure 5. The main advantage of this new model is that it is fully customizable and covers a wide range of created formulas. Additionally it would be possible to optimize the parameters of the creation model using feedback from carried out tests while the target application is fuzzed. As already mentioned this is not considered in more detail in this thesis.

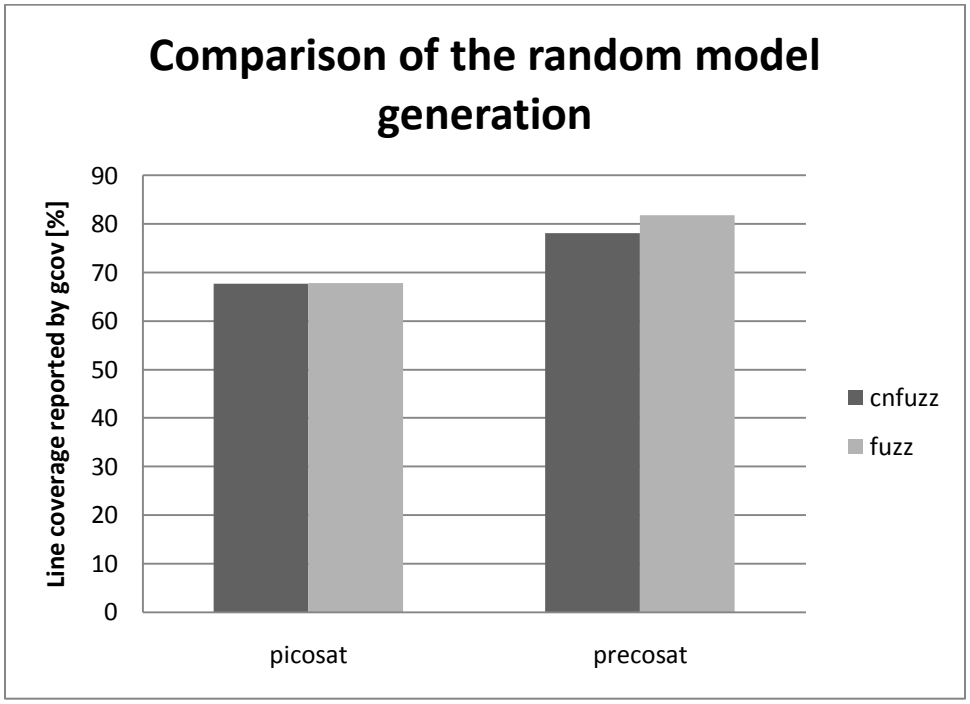

Figure 5: Comparison of the random model generation

2.2 Getting feedback from application runs

Having the output of gcov and analyzing the source code that was not tested by the fuzzer would make it possible to generate inputs that cover these parts too. But as this thesis is about black box fuzz testing and creating test cases in an automated way a mechanism to get feedback from carried out tests is necessary. That way a fuzzer can determine whether certain input data reveals newly executed parts of the tested application. Many different kinds of

information about an application run are available. In the current implementation the fuzzer retrieves information about the return value of the solver, the processing time and the trace information which are described in more detail in the next sections.

2.2.1 The return value

The return value is often used by sat solvers to indicate whether the problem is SAT, UNSAT or not calculated for some reason. The SAT solvers used in this thesis are designed to return 10 if a problem is SAT and 20 if the problem is UNSAT. A value of zero is returned if no decision could be made, e.g. the input or the arguments are invalid. In case of an abnormal termination a value according to the specifications of the runtime environment is returned. The fuzzer checks whether a certain problem is recognized as SAT or UNSAT by checking the return value of the solver. The concrete return values can be specified in a configuration file which is used by the fuzzing application where it is also possible to specify a list of accepted return values, e.g. zero for invalid input. If the return value doesn't match any of the defined values the execution is handled as abnormal and the input is considered to unveil an error.

2.2.2 The execution time

The time it takes to execute the solver given a certain input is measured and saved for each input. This value is of interest because it indicates how difficult it was to determine the satisfiability of the problem for the solver. Especially the ratio between the execution time and the size of the formula gives information about the inability of the solver to find structures in the formula and to speed up the solving process. Nevertheless this information is not used in the final version of the fuzzing application because it does not fit into the pursued strategy. The execution time is used solely to specify timeouts in the SAT solver. This is necessary to prevent inputs which block the fuzzing process for a long time and decrease the overall number of inputs that can be tested.

2.2.3 Monitoring

This is the most expensive information the fuzzer will get about a certain test run and may be a profit-yielding innovation compared to other fuzzing tools. The first attempt to get information about the executed parts of a program was to use the processor's single step debugging mode and to trace all the executed code by the values of the instruction pointer at every step. As explained in more detail in Sec. 6.6.1 this approach failed because it dramatically slows down performance when running an application in single step mode which would make it impossible to run the desired amount of tests within an appropriate amount of time. After some further investigation Intel's pin tool (Luk, et al., 2005) proved to be of great value in getting the desired information. The tool executes code a virtual machine and offers methods to inject and execute user supplied code at defined positions by providing a dynamically linked library to the tool. A test was carried out by creating a library for the tool to intersect every instruction of an

application and trace physical addresses. To be more precise it was not necessary to store every address but the address of the first instruction of every encountered basic block (BBL) (Wikipedia, 2009) which are those parts of code that have a single entrance single exit nature as depicted in Figure 6. Therefore if the first instruction of a basic block is executed all other instructions will be executed too and there is no need to store this information which greatly improves performance of the monitoring process without losing information. The data from monitoring allows generating an exact trace of an application's run within a short time as documented in Sec. 6.6.2. Although this approach seemed to be very promising at first glance it proved that running it with real data on a SAT solving application was unfeasible too. As example, running the sat solver precosat with a rather hard satisfiability problem turned out to execute about 239 million basic blocks. If just the address of every executed basic block was stored with a minimum of 4 bytes the amount of data required for a single trace would be about 911 MB as demonstrated in Sec. 6.6.3. Although providing that much storage would not be a problem, processing that many data would again significantly decrease performance. The final attempt is to abandon some information and to monitor only which of the static basic blocks were executed which provides the most suitable tradeoff between information and performance. The tested attempts are additionally described in Sec. 2.3.5 and Sec. 6.6. The optimized version finally used in this thesis will be described next.

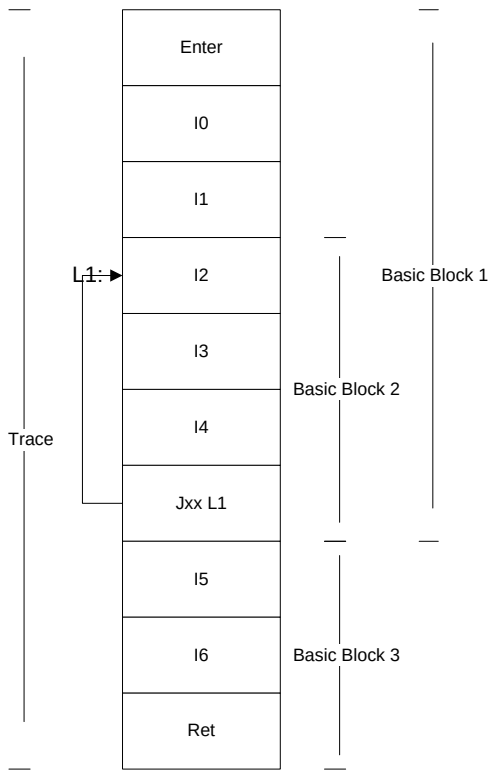

Figure 6: Basic Blocks

2.3 Description of the Monitoring tool

The Monitoring tool is implemented in the C++ programming language and must be compiled specific to the target system which can be Microsoft Windows or a Linux based operating system. Detailed instructions on building and running the tool on these systems are given in Sec. 6.2.3. The Monitoring tool generally traces three kinds of information of a target application which are the binary layout of the main image, the executed basic blocks and exceptions in case they occurred. Details about this information are given in the next sections. Information about the implementation of the monitor tool and certain code excerpts are given in more detail in Sec. 6.3.

2.3.1 Binary layout of the image

The monitoring library instructs the pin tool to intersect all binary image loads which are the loading of the main image and all dynamically linked libraries. In the final version the tool only traces data about the main image as the dynamically linked libraries are usually part of the operating system and of little use to the fuzzer. The information traced about the main image are the path to the image in the file system, the high and the low address of the image in the virtual address space and the sections the image is made of. Executable images may contain a different number of sections which is described in more detail at (Wikipedia, 2009). As the

fuzzer is only interested in code that can be executed and not the data of the application only sections which are marked as executable are reported. The monitor tool traces information about each executable section in the main image such as the name of the section, the offset in the binary image and the size of the section. With this information the fuzzer has information about the amount of executable code and can produce coverage information.

2.3.2 Executed Basic Blocks

The pin tool is instructed via the monitor library to insert code every time a new basic block (Wikipedia, 2009) is discovered. The inserted code stores basic information about the basic block, such as the address, the size and an execution counter. Additionally a small amount of code is dynamically inserted into each recognized basic block and executed every time the basic block is executed. The inserted code simply increments the execution counter stored for every recognized basic block. That way it is possible to report how often each basic block was actually executed by a certain application run. More details about the implementation are given in Sec. 6.3.

2.3.3 Assertions

Anomalies, such as invalid memory accesses which are detected by the execution environment are notified by certain mechanisms which are signals in the case of Linux based operating systems and exceptions in the case of Windows based operating systems. As the pin tool uses the debugging interface to instrument a target application these notifications are signaled to the pin tool via a context change debugging event (Intel Corporation, 2009). The monitor library instructs the pin tool to execute code of the monitor library every time this happens. This code tracks some information about the anomaly which is basically the reason for the anomaly such as a fatal signal caused by a divide by zero or a segmentation fault caused by invalid memory access. Check instructions such as assert or abort are usually implemented to raise the debugging interrupt of the system which will be detected by the same mechanism. Therefore they will also be handled and reported as assertions in case they are used and executed in the code.

2.3.4 Communication between the fuzzing application and the monitoring tool

Having a library which runs in process space of the tested application requires a method to communicate with the outside. Although many different technologies such as messages or shared memory are possible using files as buffer proved to be very simple and robust. Therefore the monitor library simply serializes all the collected information to a file. The information is then de-serialized by the fuzzing application and used for further processing. As the fuzzer is implemented using Microsoft's .Net framework technology which offers full serialization support for XML files and XML provides information in a structured and readable way it is used as format for the temporary trace information. Figure 7 depicts the basic layout

of such a trace file in the form of a class diagram and Figure 8 shows some excerpts of such a file.

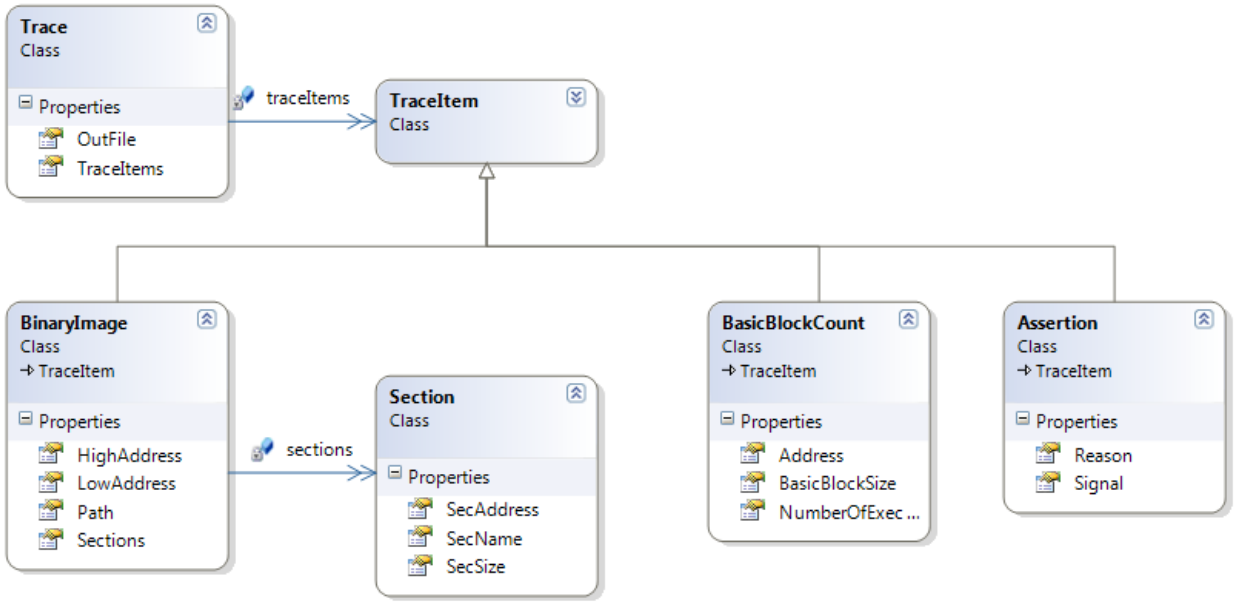

Figure 7: Class diagram of the generated trace information

```
<?xml version="1.0" encoding="utf-8"?>
<Trace     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
OutFile="trace.xml">
        <TraceItems>
                <TraceItem xsi:type="BinaryImage">
                        <Path>/home/juergen/Documents/bin-picosat/picosat</Path>
                        <LowAddress>134512640</LowAddress>
                        <HighAddress>134655455</HighAddress>
                        <Sections>
                                <Section>
                                        <SecName>.init</SecName>
                                        <SecAddress>2808</SecAddress>
                                        <SecSize>48</SecSize>
                                </Section>
                                <Section>
                                        <SecName>.plt</SecName>
                                        <SecAddress>2856</SecAddress>
                                        <SecSize>704</SecSize>
                                </Section>
                                <Section>
                                        <SecName>.text</SecName>
                                        <SecAddress>3568</SecAddress>
                                        <SecSize>100636</SecSize>
                                </Section>
                                <Section>
                                        <SecName>.fini</SecName>
                                        <SecAddress>104204</SecAddress>
                                        <SecSize>28</SecSize>
                                </Section>
                        </Sections>
                </TraceItem>
                <TraceItem xsi:type="BasicBlockCount">
                        <Address>3568</Address>
                        <BasicBlockSize>33</BasicBlockSize>
                        <NumberOfExecutions>1</NumberOfExecutions>
                </TraceItem>
                <TraceItem xsi:type="BasicBlockCount">
                        <Address>3128</Address>
                        <BasicBlockSize>6</BasicBlockSize>
                        <NumberOfExecutions>1</NumberOfExecutions>
                </TraceItem>
    :
    :
    :
        </TraceItems>
</Trace>
```

Figure 8: Excerpts of a trace file

2.3.5 Performance of Monitoring

Pin requires a user supplied library which specifies the intersection points and contains all the methods which are to be called at these intersection points. Sec. 6.3 describes these mechanisms in more detail. In early phases of the development a full featured version of the monitor tool was used to test many behaviors of pin in more detail. Table 5 and Table 6 compare the performances of various libraries used with the pin tool at these early phases. The table shows the number of tests that could be executed within 15 min using cnfuzz as formula generator and picosat as well as precosat as solver. The cnfuzz tool was seeded with a predefined set of numbers and created the same set of formulas for every test. Inscount2 is delivered as an example tool with the pin installation package and simply counts the number of executed basic blocks. The initial test with the monitor tool showed that the number of tests which could be performed within the testing time is significantly below the number of tests possible with inscount2. In the next attempt all parts of a binary which belong to OS specific code were omitted by skipping tracing information of loaded libraries. This way only code of the main image was instrumented which raised the performance to around the level of the

inscount2 tool. Finally a downgraded version of the tool was tested which collects only required information as described in the previous section and is the version of the tool that will be used throughout this thesis. Although performance of the tool could be raised by 50% compared to the initial version there is still a penalty of factor 20 speed compared to running the solver without monitoring.

Performance of Monitor Tool – picosat			
Time	**15 min**		
Solver	**Picosat**		
	num tests	time/test [s]	
no tool	15935	0.056	100%
inscount2	644	1.398	2474%
monitor full featured	494	1.822	3226%
monitor full featured main image only	677	1.329	2354%
monitor optimized	778	1.157	2048%

Table 5: Performance of Monitor Tool - picosat

Performance of Monitor Tool – precosat			
Time	**15 min**		
Solver	**Precosat**		
	num tests	time/test [s]	
no tool	13658	0.066	117%
inscount2	536	1.679	2973%
monitor full featured	442	2.036	3605%
monitor full featured main image only	564	1.596	2825%
monitor optimized	660	1.364	2414%

Table 6: Performance of Monitor Tool - precosat

2.4 The fuzz application

The next sections describe the fuzz application which implements the random creation model described in Sec. 2.1 and uses feedback from application runs to direct the fuzzing process. The application is implemented in the C# programming language using Microsoft's .Net Framework and is compiled with Microsoft's C# compiler. The application runs natively on Windows operating systems and requires the mono runtime to execute on Linux based systems. Because the generated CIL code (Wikipedia, 2009) is binary compatible across these operating systems no recompilation is necessary to run the fuzzer on Linux based systems. Detailed instructions on installing the fuzzing application as well as setting up the runtime environment is given in Sec. 6.2.

2.5 The application main loop

The main loop of the fuzzing application is depicted in Figure 9. The core task of the application after initialization is to repeatedly create a new input, execute the target application with that

input, get execution information and update the internal state as well as provide statistics and visualizations of the fuzzing progress. In case the tool is terminated it will generate final statistics as well as a test suite with inputs that cover all the code locations that were executed by the fuzzer. If exceptional behaviors, such as assertions, invalid return values or long execution times are detected by the fuzzer the corresponding inputs will also be provided as part of the test suite. The following sections describe every step of the main loop of the fuzzer in greater detail.

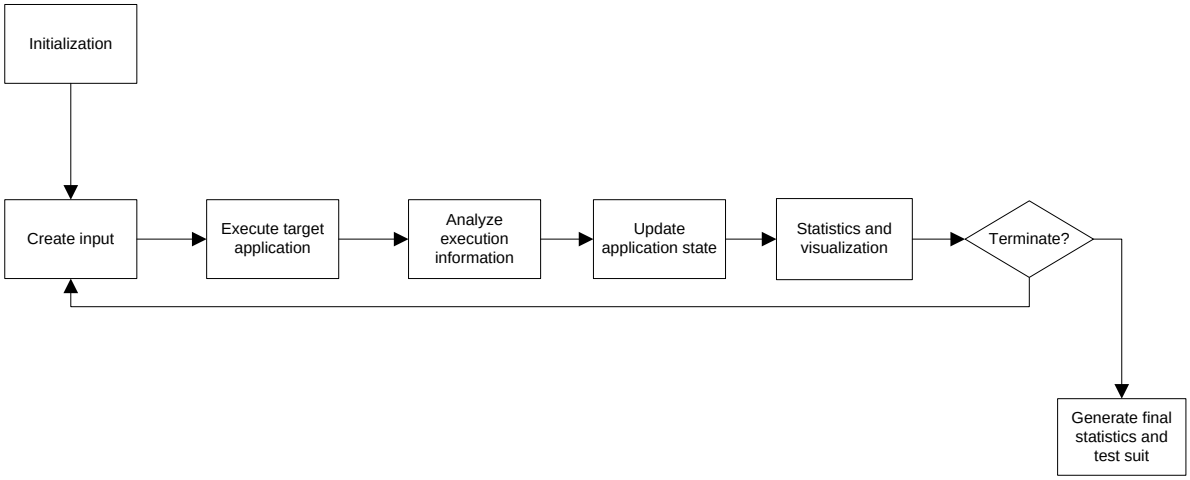

Figure 9: Main loop of the fuzzer

2.5.1 Initialization of the fuzzer

In this step the internal state of the fuzzer is set up. This mainly involves analysis of the arguments to the tool and all the settings in the configuration file used by the fuzzer. Both are described in more detail in Sec. 6.1.

2.5.2 Create input

If the loop is entered the first time an initial input is created as described in Sec. 2.1. At further runs the input is created corresponding to the information collected in previous runs. Possibilities are to create another input reusing the random input creation model or to choose an existing input and mutate it. Selection of inputs is described in detail in Sec. 2.8 and mutation of these inputs is described in detail in Sec. 2.7. As described in Sec. 2.17 the fuzzer also supports generation of invalid inputs and fuzzing of arguments which is handled in this step too.

2.5.3 Execute target application

In this step the solver application is started via the pin tool using the monitor library. Arguments are passed to the solver if specified and the input formula is serialized to a file if this is required by the arguments. The input formula is then sent to the solver via the standard input stream. The actual sending of the input is done asynchronously by a separate thread. This is

necessary to prevent a deadlock in the fuzzer in case the target application does not accept input data, e.g. due to a crash. If the target application does not quit within a certain amount of time, passed as another argument to the fuzzing application, it will be terminated which is done in two phases. In the first phase the target application will be closed by rising the terminate signal and the application has the possibility to shut down properly. If this does not succeed within a certain amount of time the application will be closed by killing the corresponding process. The drawback of the second method is that the application as well as the monitor tool which runs in the address space of the application will not finish properly and therefore no valid trace will be generated. The details of closing an application are specific to the operating system and will be given in Sec. 6.4.

2.5.4 Analyze execution information

In this step the generated information, as described in Sec. 2.2, will be analyzed. This involves obtaining the return value and storing the amount of time it took to execute the application or specifying a flag in case the application was closed by the fuzzer because of exceeding the available processing time. The execution trace is de-serialized using the XML serialization methods of Microsoft's .Net framework (Microsoft Corporation, 2009). This operation results in a tree of objects that correspond to the class diagram depicted in Figure 7.

2.5.5 Update application state

The information gathered in the previous step is used to update the state of the fuzzing application. This mainly involves traversing the object tree representing the application trace and updating the corresponding coverage information. Details about the stored information and the update process are given in Sec. 2.6.

2.5.6 Statistics and visualization

To give response about the current fuzzing progress several statistics such as the current amount of covered code as well as information about the used inputs and mutators is printed to the console. Additionally the progress of the fuzzing tool is visualized as depicted in Figure 10. The form displays the execution status of the basic blocks in an image and the amount of covered code in a progress bar. Every pixel in the image represents one address of the main image of the target application and is colored according to the execution state of the address. The meaning of the different colors is described in more detail in Table 7.

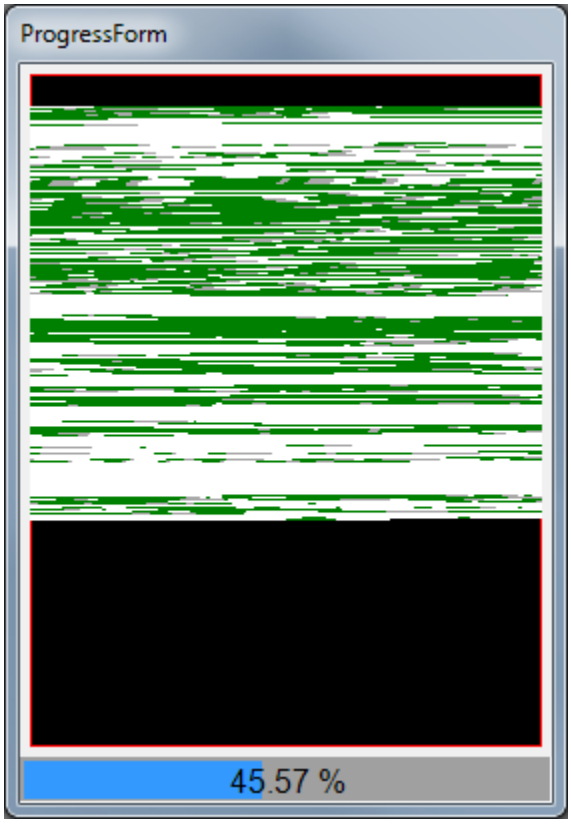

Figure 10: Visualization of the fuzzing progress

Color	Meaning
Black	The address belongs to the binary main image
White	The address is within an executable section
Dark Gray	The address was recognized as executable code
Green	The address was executed by one of the tests

Table 7: Colors of the visualization form

2.5.7 Generate final statistics and test suite

The main loop of the fuzzing application can be terminated by user input or if the testing time exceeds a certain duration which can be specified by an argument to the application. In both cases the fuzzing application generates a number of files in a specified output directory which includes statistics of the carried out tests and optionally a number of input files which caused anomalies in the tested application. Additionally a test suite is generated which covers all the basic blocks that were detected and executed during the overall fuzzing progress. Details about the output and the test suite are given in Sec. 6.9.

2.6 Managing the state of the fuzzing progress

The following sections describe how information gathered by executing the target application is stored within the fuzzing application. Before these details are given the next section highlights a certain problem that must be handled due the way the pin tool detects basic blocks.

2.6.1 Problem of overlapping basic blocks

Basic blocks may overlap due to the way they are recognized by the pin tool as depicted in Figure 11. A new basic block is generated every time the CPU starts working after a control instruction has been processed. The basic block ends whenever another control instruction is encountered. If the target of further jump instructions is within an already detected basic block a new basic block is generated although it overlaps with the existing one. To get accurate information about coverage further attention has to be given to these situations. A simple and effective approach is to store coverage information per basic block and to have a reference to the basic block at every address it contains. The other way around every address will store a list of all the basic blocks that have been detected which contain the address. The additionally required memory is not a real problem as the amount of available addresses as well as static basic blocks is constant. Therefore the amount of additionally required memory will be constant too.

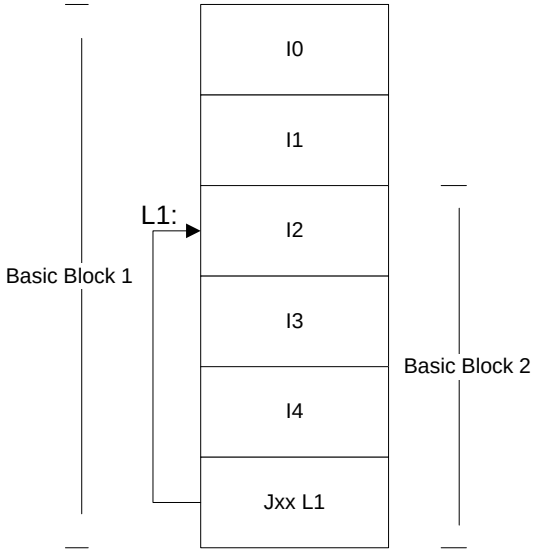

Figure 11: Overlapping basic blocks

2.6.2 Managing Coverage information

Coverage information is tracked in the fuzzer in address arrays grouped by sections of the binary image. They are generated if the target application is executed for the first time. Every used input is stored in an input cache and a reference to the input is stored at every basic block that has been executed or was recognized by the input. Every address that is contained in the basic block stores a reference to it as described in the previous section. The overall layout is depicted in Figure 12. This way all the relevant information generated by the monitor tool is available to the fuzzing application. The drawback of storing all the information is the linear

grow of memory consumption as depicted in Figure 13, which is caused by storing all the inputs. This limits the total number of tests that can be carried out before performance drops drastically which is caused by system load. On the test environment, as described in Sec. 6.8, this happens after about 1100 tests when the total amount of consumed memory reaches nearly 700 MB. Nevertheless, throughout the tests carried out in this thesis the full information is stored and used to get accurate information about the behavior of different strategies. Sec. 2.13 later in this thesis describes a strategy to limit overall memory consumption by dropping some of the information which makes it possible to use the fuzzer for an unlimited number of tests.

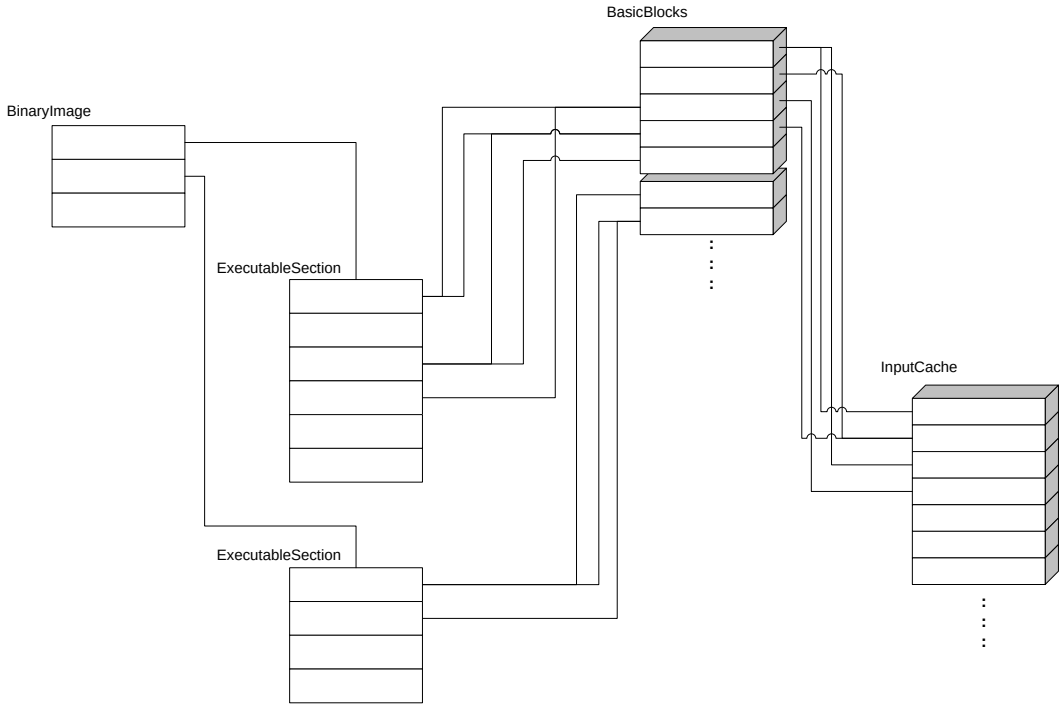

Figure 12: Layout of the stored coverage information

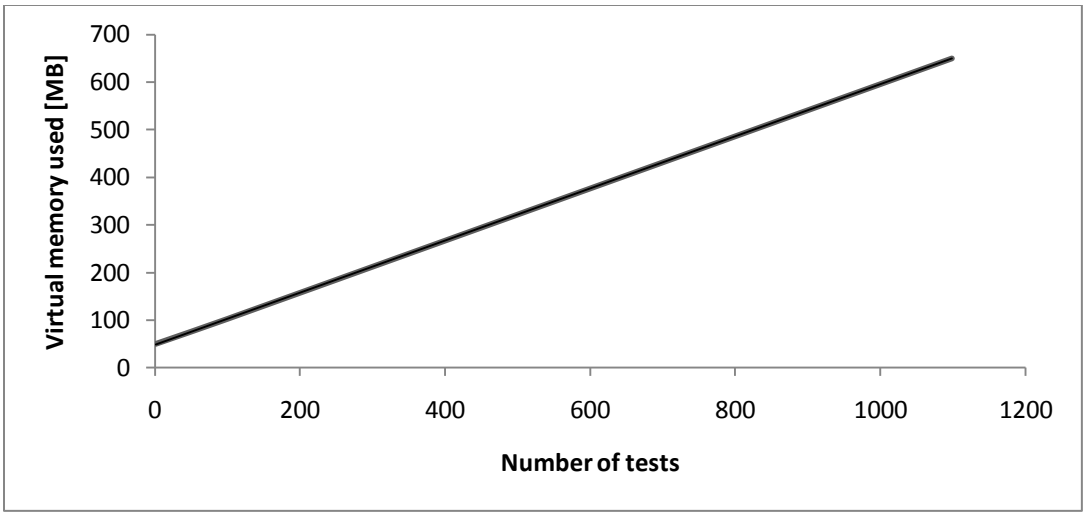

Figure 13: Memory consumed by coverage information

2.6.3 Usage of the input cache

As described in the previous section references to all inputs are additionally stored in the input cache which is for two reasons. The first reason is that an input which has already been processes can be used by mutators to generate a new input. The second reason is that prior to monitoring an application it is checked if the input has already been used. This may happen especially if very short formulas are generated by the mutation progress. The basic idea behind preventing a certain input to be monitored multiple times is that this step is, by far, the most time consuming one. The drawback of this strategy is that storing all inputs constantly consumes memory as described previously. Detailed analysis of the effect of caching inputs in Sec. 2.11 show that the gain in the number of tests carried out is negligible and can be compensated by increasing testing time.

2.6.4 Calculation of coverage by the fuzzer

This is the first time information from application runs is used. Nevertheless it is not used to influence the fuzzing process but to get basic information about the amount of code coverage while the target application is fuzzed by calculating the ratio between the number of executed code bytes and the total number of executable code bytes given by the sum of the sizes of all executable code sections which is also shown by the formula below. The gained coverage information is also visualized as described in Figure 10. There is a remarkable difference between the coverage values determined this way compared to the line coverage value determined by tools such as gcov which is also shown in Table 8. This seems plausible as there is no direct match between the number of code lines and the number of generated code bytes. It also seems plausible that the amount of byte coverage is lower than the amount of line coverage as there is a remarkable amount of automatically generated code in every executable.

$$byte\ coverage = \frac{Num\ executed\ code\ bytes}{\sum sizes\ of\ executable\ sections}$$

coverage analysis by the fuzzer				
SAT solver	**time**	**num tests**	**byte coverage**	**line coverage**
picosat	15 min	411	59.71	67.46
picosat	60 min	943	59.76	67.46
precosat	15 min	347	66.74	81.42
precosat	60 min	959	67.16	81.52

Table 8: Byte coverage determined by the fuzzing application

2.7 Mutation of already performed tests

The next sections describe mutation-based strategies implemented in the fuzzing application in order to slightly change existing inputs to generate new ones. During the following sections only mutators that create valid formulas are used. That way it is possible to compare the results with the cnfuzz tool which solely creates valid formulas too. Later in this thesis additional mutators are described which generate invalid formulas or operate on the arguments to the SAT solver under test to further increase coverage.

2.7.1 Coverage achieved by using mutators only

To get information about the behavior of mutators a special test setup was implemented which starts with an initial random input and generates all further inputs by mutation of existing inputs only. For each group of mutators a number of tests were carried out to get suitable configurations for each mutator experimentally. Although it might be possible to get these values during the fuzzing of the target application it was not implemented in this thesis. Besides the high complexity, the main reason is that it would dramatically increase the number of parameters in the whole system and therefore the overall number of tests required to achieve high code coverage would be further increased. The following sections give information about the implemented mutators and the tests carried out to experimentally find suitable values for their parameters. As already mentioned all tests use the same formula as starting point. This formula consists of 58 variables and 781 clauses and is satisfiable. The gcov tool reported that this input covered 42.16% of the executable bytes and 53.40% of the code lines of picosat.

2.7.2 RandomNew

This mutator creates a completely new formula as described in Sec. 2.1. In the tests described in this section it is used solely to create the initial input. In the final version of the fuzzing application this mutator will be used randomly according to the state of the fuzzer.

2.7.3 AddClause / DeleteClause

The mutators AddClause and DeleteClause are used to modify clauses. If clauses are added they will be created with the same method that was used to generate random input formulas as

described in Sec. 2.1. If clauses are deleted they will be chosen randomly among existing clauses. The mutators expect the number of add or delete operations as parameters which are chosen randomly from an exponential distribution. A few tests showed that it is relevant whether the number of different variables in the clause is adapted based on the number of clauses the formula will contain after the mutation operation. If the number of different variables is adapted the mutated formula tends to have the same satisfiability as the randomly generated formula. The reason may be that creating bigger formulas with equal density does not impact the satisfiability of the formula very much. Not updating the number of variables makes a formula more satisfiable if clauses are removed and less satisfiable if clauses are added. The drawback is that all formulas will have the same number of variables which is equal to the number of variables in the initially randomly generated formula. This will omit many possible configurations of formulas. Therefore a flag which decides whether the number of variables should be adapted is required as additional parameter by the mutators. Table 9 and Table 10 list some tests which were carried out in order to get suitable values for parameters experimentally.

Solver	**Picosat**						
Time	**5 min**						
Mutators	**AddClause + DeleteClause**						
Tests							
Add Mean	**Del Mean**	**Total**	**Sat**	**Unsat**	**TimeOut**	**byte coverage**	**line coverage**
1000	1000	223	41	182	0	54.82	64.36139762
500	500	230	99	131	0	51.59	62.0231011
2000	2000	193	12	180	1	51.34	62.0231011
1500	1500	225	35	190	0	54.91	64.36139762
1250	1250	222	43	179	0	54.23	64.01437128

Table 9: Optimize AddClause/DeleteClause - picosat

Solver	**precosat**						
Time	**5 min**						
Mutators	**AddClause + DeleteClause**						
Tests							
Add Mean	**Del Mean**	**Total**	**Sat**	**Unsat**	**TimeOut**	**byte coverage**	**line coverage**
1000	1000	202	118	84	0	57.1	65.16286321
500	500	218	170	48	0	54.19	63.97305862
1500	1500	196	155	41	0	56.8	64.95629992
750	750	205	161	44	0	57.28	65.4190017

Table 10: Optimize AddClause/DeleteClause - precosat

2.7.4 SwapSign

The SwapSign mutator randomly selects variables in a formula and swaps their sign. The mutator takes the number of variables to be swapped as a parameter which is chosen

exponentially distributed. Table 11 shows some tests that were carried out in order to get a suitable value for this parameter experimentally. The tests also show that this mutator does not have great impact on the line coverage of the target application. In the tests depicted it was not even possible to transform the initially satisfiable formula into an unsatisfiable one.

Solver	**picosat**					
Time	**5 min**					
Mutators	**SwapSign**					
Tests						
Exp Mean	**Total**	**Sat**	**Unsat**	**TimeOut**	**byte coverage**	**line coverage**
1000	227	227	0	0	48.88	60.45
2000	226	226	0	0	48.87	60.20
500	228	228	0	0	49.15	60.50
250	229	229	0	0	49.06	60.24
400	224	224	0	0	49.03	60.50
750	225	225	0	0	49.08	60.50

Table 11: Optimize SwapSign

2.7.5 SwapVar

This mutator changes variables in a formula. Again the number of operations is chosen from an exponential distribution which is given as parameter to the mutator. The new value for a variable is chosen from a Gaussian distribution with the old value of the variable as mean. The deviation of the Gaussian distribution is additionally required as parameter by the mutator. Table 12 shows a number of tests that were carried out in order to find suitable values for the parameters to the mutator experimentally. Similar to the SwapSign operator the tests showed that the amount of covered code lines could not be increased very much by this mutator.

Solver	picosat						
Time	5 min						
Mutators	SwapVar						
Tests							
Exp Mean	Deviation	Total	Sat	Unsat	TimeOut	byte coverage	line coverage
1000	200	231	231	0	0	47.12	56.11654206
100	10	233	233	0	0	46.78	56.11654206
500	10	234	234	0	0	48.83	59.76858114
1000	10	220	220	0	0	46.6	55.18287596
750	10	230	230	0	0	47.15	56.11654206
300	10	231	231	0	0	46.77	56.0752294
500	100	226	226	0	0	49.67	60.57830926
500	200	228	228	0	0	49.57	60.32217077
500	150	228	228	0	0	49.56	60.32217077
500	75	230	230	0	0	46.8	56.0752294

Table 12: Optimize SwapVar

2.7.6 AddLiteral / DeleteLiteral

These mutators change individual clauses of formulas by adding or removing a literal. The mutator expects the number of clauses that should be changed as parameter which is chosen from an exponential distribution. The particular clauses that are modified are selected equally distributed among all clauses in the formula. If literals are added their value and sign will be chosen uniquely distributed among the possible values. It is again possible to adapt the number of different variables of the clause which is specified as additional argument to the mutator. This has an effect if e.g. all occurrences of the highest variable of the formula are deleted. Table 13 lists some tests carried out in order to get suitable values for the parameters experimentally.

Solver	picosat						
Time	5 min						
Mutators	AddLiteral + DeleteLiteral						
Tests							
Add Mean	Del Mean	Total	Sat	Unsat	TimeOut	byte coverage	line coverage
100	100	232	155	77	0	51.53	62.10572642
500	500	245	126	119	0	51.93	62.23792693
1000	1000	257	83	174	0	50.09	60.83329652
750	750	262	88	174	0	51.36	61.72564996
300	300	252	116	136	0	51.78	62.26271453
500	300	255	114	141	0	51.17	61.17206032

Table 13: Optimize AddLiteral/DeleteLiteral

2.7.7 Conclusion of the mutator tests

To compare the benefit of different strategies tested in this thesis with the strategy of not using any feedback as implemented by the cnfuzz tool these mutators are the only one used by now. Additional mutators that will modify the generated input stream and the arguments passed to the target application will be introduced later in Sec. 2.17. The tests carried out in the previous sections show that the overall coverage can be influenced by these mutators and that every group of mutators has its own capabilities. Changing the available literals or clauses in the formula made it possible to cover a significant amount of new code and to generate unsatisfiable formulas from the initially satisfiable one. SwapVar and SwapSign changed the characteristics of the initial input only slightly which implies that they highly depend on the initial input. This behavior may be useful in cases where it is necessary to change a suitable input formula only minimal to cover a certain area in the application code. It is also important to note that the tests are used solely to get initial values for the parameters of the mutators. As there may be high correlations between the parameters of the mutators it will be necessary to further optimize them when they are used all at once in the final fuzzing application which will partly be done in Sec. 2.14.

2.7.8 Coverage with these mutators

At this time a special test is carried out which randomly chooses any of the above mutators and uses the parameters found experimentally by the tests in the previous sections. Table 14 and Table 15 give some details about the tests using picosat and precosat as SAT solving applications and running them for 15 minutes each. Until now no feedback of the tested application is used. Therefore the results in the table can be used to check the benefit of using feedback to select a certain mutation strategy. At this point it is no surprise that using a randomly chosen mutator that operates on a completely randomly selected input does not increase the amount of covered code compared to the attempt of using the random creation model solely. The advantage of using mutators to create further inputs will be given when suitable inputs and mutators are selected by certain strategies that use feedback from application runs.

Solver	**Picosat**	
Time	**15 min**	
Mutators	**All randomly**	
Test		
num tests	**byte coverage**	**line coverage**
679	57.16	66.57

Table 14: All mutators randomly - picosat

Solver	Precosat	
Time	15 min	
Mutators	All randomly	
Test		
num tests	byte coverage	line coverage
575	64.1	79.86

Table 15: All mutators randomly - precosat

2.8 Using feedback from test runs

Basically there are two positions in the fuzzing process where feedback can be used. The first one is when the mutator is selected and the second one is when the input that the mutator will use is selected. The next sections describe several strategies that were implemented and tested to use feedback at these two positions.

2.8.1 Using feedback to choose mutators

The process of selecting a mutator based on feedback from application runs used in this thesis is rather simple and therefore it is considered first. The fuzzer tracks statistics about each available mutator in the application state whereas a special fitness value field is included. Every time a mutator is used successfully the fitness value will be increased by a certain amount. Every time a mutator is selected to modify some input the fitness value will be. The mutator to be used next will be selected based on the current fitness values of all the mutators. Details about this selection process will be given in the next sections.

2.8.2 Details of the mutator selection process

To get information about the suitability of mutators statistics about them are collected during tests. These statistics include how often a certain mutator, including its parameters, is used and how often the mutator succeeded in generating a suitable new input. A new input is considered suitable if it executes code which has not been executed by previous runs or if basic blocks are newly recognized by the pin tool, even if these basic blocks have not been executed by this input. The main purpose of this statistics is to check the quality of the selection process, e.g. how many times was a certain mutator chosen and how many times did it yield a successful mutation. As already mentioned a special fitness value is additionally stored for each mutator which reflects the current suitability of the mutator. This value is incremented every time the mutator was used successfully by a certain value which can be specified in a configuration file used by the fuzzing application. Suitable values for the SAT solvers picosat and precosat will be optimized experimentally throughout this and the following sections. Every time the mutator is chosen via the selection algorithm the fitness value is decremented whereas the value is prevented to get smaller than one. Mutators are chosen according to their fitness value whereas two different selection algorithms will be tested in the following sections which are proportionate selection and tournament selection (Bäck, 1996).

2.8.3 Proportionate selection

In this scenario mutators will be selected proportional to their fitness value. Therefore the probability of a certain mutator of being selected can be given by the formula depicted below. The selection process is implemented by choosing a random number out of the sum of all the fitness values. The mutator selected is the one whose fitness range surrounds the value chosen if the fitness values of the mutators are arranged as depicted in Figure 14.

$$p_{m_i} = \frac{f_{m_i}}{\sum_j f_{m_j}}$$

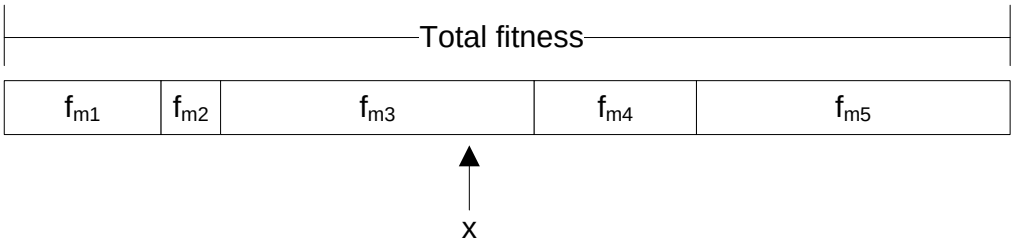

Figure 14: Proportionate selection

2.8.4 Tests with mutator fitness using proportionate selection

The following tables show some tests that were carried out in order to find suitable values for the parameters to calculating the mutator fitness when proportionate selection is used. Table 16 gives details of using the picosat solver and Table 17 gives details of using the precosat solver.

Solver		**picosat**				
Time		**15 min**				
Mutator selection		**Proportionate selection**				
Tests						
Bonus	**Total**	**Sat**	**Unsat**	**TimeOut**	**byte coverage**	**line coverage**
3	676	417	259	0	59.28	66.98888275
10	661	483	178	0	59.02	67.03019541
50	665	552	113	0	58.23	67.03019541
5	676	334	336	6	58.76	66.98888275
15	654	505	149	0	59.69	67.6746729
20	663	553	110	0	59.11	67.03019541

Table 16: Optimizing proportionate selection - picosat

Solver		precosat				
Time		15 min				
Mutator selection		Proportionate selection				
Tests						
Bonus	Total	Sat	Unsat	TimeOut	byte coverage	line coverage
15	530	452	78	0	65.85	80.84395147
10	545	415	130	0	66.39	81.24264035
20	541	463	78	0	65.87	80.38459254

Table 17: Optimizing proportionate selection - precosat

2.8.5 Tournament selection

In this scenario a number of n mutators are chosen randomly to participate in a tournament. The mutator which will be selected is the winner of the tournament which is the one with the highest fitness value. The selection strategy is also depicted in Figure 15. The advantage of this selection method is that the size of the tournament can be used to easily adjust the selection pressure. If the tournament size equals one the selection strategy is equal to random selection. The greater the size of the tournament the more likely the mutator with the overall highest fitness value will be chosen. A tournament selection with n participants can be implemented very easily by randomly choosing a mutator n times and remembering the most suited encountered mutator. The implementation does not prevent selecting the same mutator multiple times. This has the benefit that also any of the worst n-1 mutators can win the tournament if worse mutators are chosen more than once. This is not possible if multiple occurrences of the same mutator in the tournament are not allowed. The following section gives detail of a number of tests that were carried out in order to get suitable values for the fitness increment as well as the tournament size when tournament selection is used.

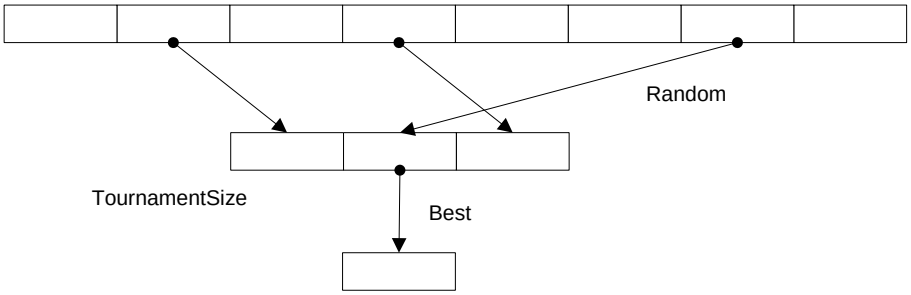

Figure 15: Tournament selection

2.8.6 Tests with mutator fitness and tournament selection

Table 18 describes tests that were carried out in order to find suitable values for the parameters to calculating the mutator fitness and the optimal size of the tournament when tournament selection is used to select a mutator. As the tests yielded worse results for the tests of the picosat solver the tests of the precosat solver are omitted. Although it may be possible to

further adjust the parameters to have equal results as with proportionate selection the final version of the fuzzing application will use proportionate selection due to reasons explained in the next section.

Solver		Picosat					
Time		15 min					
Mutator selection		Tournament selection					
Tests							
Bonus	TournamentSize	Total	Sat	Unsat	TimeOut	byte coverage	line coverage
15	3	672	458	214	0	58.71	66.94757009
15	10	619	232	385	2	57.2	66.56749363
10	5	672	334	338	0	58.59	66.94757009
10	3	670	441	229	0	58.71	66.98888275

Table 18: Optimizing tournament selection

2.8.7 Conclusion of using feedback to select mutators

The tests in the previous sections showed that the overall line coverage could be increased to 67.67% when the picosat solver was tested and mutators were chosen based on their fitness value. Compared to the tests in Sec. 2.7.8, the line coverage could be increased by 1.10 percentage points which corresponds to 26 code lines. The tests using the precosat solver showed that the line coverage could be increased to 81.24%, which is equal to 38 lines of additionally executed code. The gain in coverage is also depicted in Figure 16. To get these values the increment of the fitness value was chosen differently for each of the tested solving applications. This means that no universal suitable value is available to get optimal results although the coverage achieved with precosat is only slightly lower if the same increment value of the optimal test of picosat is used. Therefore it can be expected that the same amount of covered lines with the values used when testing picosat will be achieved if more time is given to the fuzzing process. The tests also showed that using tournament selection yielded no improvements and therefore the final version of the fuzzing application will use proportionate selection based on the assumptions of Occam's razor which says that "All things being equal, the simplest solution tends to be the best one" (Blumer, Ehrenfeucht, Haussler, & Warmuth, 1990). The next sections describe how similar methods are implemented to select a suitable input to be used by mutators.

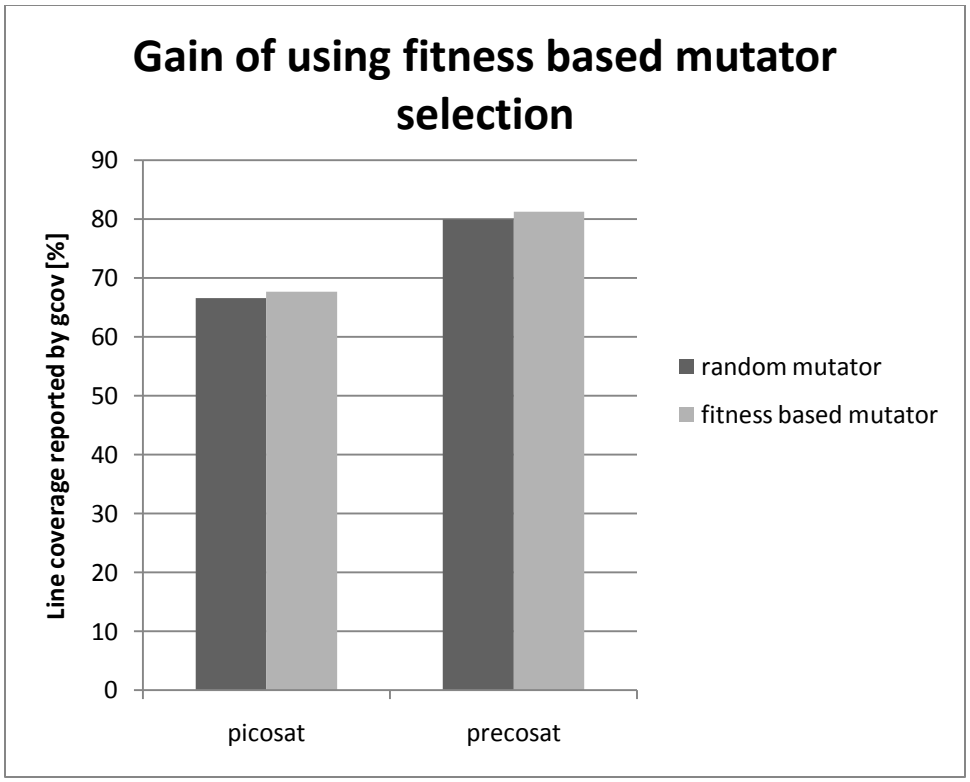

Figure 16: Gain of using fitness based mutator selection

2.9 Choosing the input to be mutated

This is the second position where feedback is used to direct the fuzzing process. Three different strategies will be tested throughout this and the following sections to estimate the benefit of using feedback to select inputs which are mutated. The most intuitive way of choosing an input is to calculate a certain fitness value for each processed input and choose one with a high fitness value. As it is the case when mutators are chosen based on their fitness value it is again possible to use proportionate selection or tournament based selection to perform this step. Because the amount of available inputs may get very large at later phases in the fuzzing progress and the costs of proportionate selection depend on the amount of available objects the primarily used strategy in the following sections will be tournament selection. Additionally tournament selection has the advantage that the size of the tournament can be used to easily adjust the selection pressure which is a valuable instrument when choosing from a fair quantity of objects. Details about the tournament selection strategy were already given in Sec. 2.8.5 when mutators were chosen based on this strategy. The following sections describe several strategies tested to find a suitable method to calculate the fitness of inputs.

2.9.1 Calculating the fitness of the input

Tournament selection requires calculating the fitness value of the available inputs in order to choose promising ones. Two different strategies will be examined throughout the following

sections to calculate this value. The first one is very similar to the method used to calculate the fitness value of mutators. The second one uses more complex strategy and considers different aspects of the input like the number of newly executed basic blocks. Another strategy will be tested which uses a completely different approach and chooses primarily suitable basic blocks to get an input.

2.9.2 Calculate the fitness of inputs based on fitness increments

This method is very similar to the way the fitness value was calculated to estimate the fitness of mutators. Basically two inputs are involved in the selection and mutation process. The first one is the input which is used by the mutator and will be referred to as base input. This input has already been used as input of the application. The second input is the one which is generated from the base input by mutation. Several statistics such as the number of times an input is chosen as base input and the number of times an input has been used successfully are stored per input. They are considered as successful if they execute code that has not been executed before or if pin succeeded in detecting new basic blocks with an input. If this is the case the number of successes of both, the currently used input and the base input, are incremented. These two statistical values are primarily used during the development phase to check whether a certain strategy leads to useful results, e.g. inputs which are chosen often should yield more successful inputs than others. Additionally a fitness value is stored per input and this is the value that is crucial in the selection process. Every time an input has been used successfully the corresponding fitness value is incremented whereas again both fitness values, the value of the currently used input and the value of the base input are updated. The actual value of the increment is tested independently for both inputs. An additional parameter will be used and tested which specifies the maximum fitness value that can be reached by a single input. This is necessary to prevent the generation of so called super individuals. These are inputs whose fitness value is drastically higher than the value of other inputs and causes them to be selected over and over again although they yield no profit. This happens at the initial phases of the fuzzing progress as the first inputs are very likely to be successful and will get a high fitness value if they are selected several times. Every time an input is selected as base input the fitness value of the input is decremented whereas the value is prevented by the algorithm to fall below zero. The following section lists some tests that were carried out in order to get suitable values for the tournament size and the increment of the fitness values experimentally.

2.9.3 Testing the fitness based input selection method

Table 19 and Table 20 show some tests which were carried out in order to find suitable values for the input selection strategy as explained in the previous section. The mutator for the tests was chosen completely random to compare the experimental values with the values found in Table 14 and Table 15. The tests show that selecting a suitable input raises the amount of covered lines to 67.29% compared to 66.57% without this strategy when the picosat solver is

tested. The additional amount of 0.72 percentage points corresponds to 17 additional code lines that were executed throughout the tests. The tests with the precosat solver showed that 81.31% of the available code lines were executed. The additional amount of 1.45 percentage points corresponds to 40 additional lines of code. This is also depicted in Figure 17.

Solver		**picosat**							
Time		**15 min**							
Mutator selection		**randomly**							
Input selection		**Tournament with fitness increment**							
Tests									
TournamentSize	**BaseInputBonus**	**InputBonus**	**Total**	**Sat**	**Unsat**	**TimeOut**	**Undef**	**byte coverage**	**line coverage**
15	5	5	613	363	250	0	0	58.65	66.94757009
15	10	5	619	336	283	0	0	58.69	66.94757009
15	5	10	604	343	261	0	0	58.95	67.28566695
20	5	5	486	284	202	0	0	58.88	66.9882158

Table 19: Fitness based input selection - picosat

Solver		**precosat**							
Time		**15 min**							
Mutator selection		**randomly**							
Input selection		**Tournament with fitness increment**							
Tests									
TournamentSize	**BaseInputBonus**	**InputBonus**	**Total**	**Sat**	**Unsat**	**TimeOut**	**Undef**	**byte coverage**	**line coverage**
15	5	10	434	256	168	10	0	64.77	79.78655922
20	5	10	461	297	161	0	3	66.63	81.31197754
30	5	10	547	325	222	0	0	63.92	79.68255342
20	10	10	550	345	205	0	0	65.88	80.38459254

Table 20: Fitness based input selection - precosat

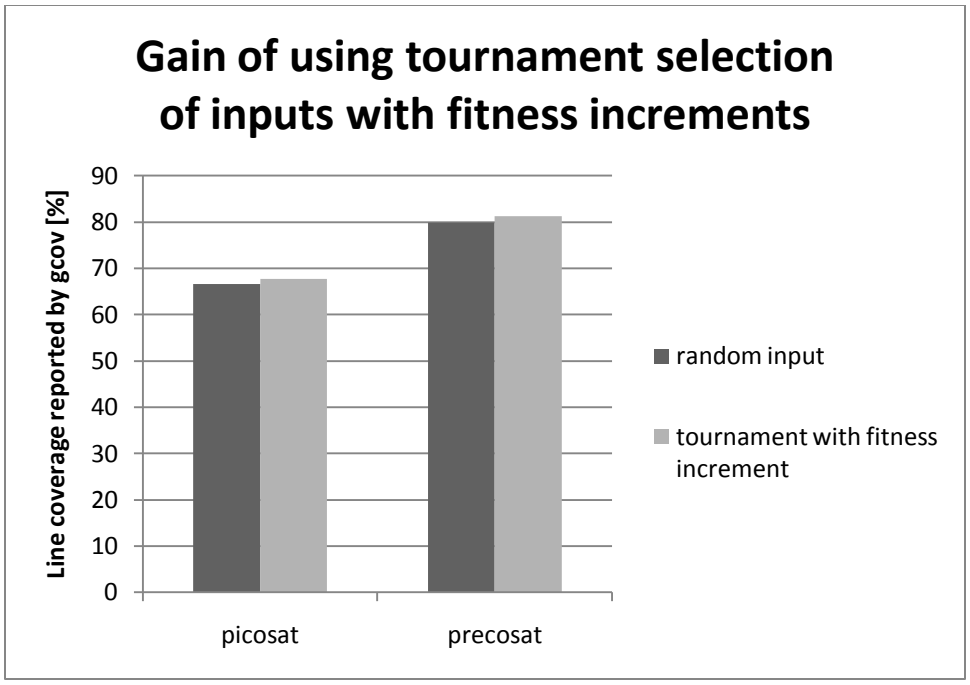

Figure 17: Gain of using tournament selection of inputs with fitness increments

2.9.4 More complex strategy to estimate the fitness value

The strategy to calculate the fitness value described in this section uses data which is more specialized to the problem of maximizing code coverage. The basic idea is that inputs which discover more basic blocks should get a higher fitness value. Inputs which discover new basic blocks at a later phase in the fuzzing process should have a higher fitness value too. Therefore fitness is calculated depending on the number of discovered basic blocks and the test number when these basic blocks where discovered. The exact calculation method is depicted in the formulas below whereas $bbls$ indicates the number of newly covered basic blocks and gen indicates the current number of tests carried out. f_{bbl} and f_{gen} are factors used to bias these values. The maximal fitness is defined by an additional parameter to prevent super individuals. Every time an input is chosen the fitness is decremented by a constant value dec.

$$fitness = bbls\ x\ f_{bbl} + gen\ x\ f_{gen}\ x\ \min(bbls, 1)$$

$$fitness_{new} = \max(0, fitness_{old} - dec)$$

A promising input is chosen again by a tournament selection with a configurable number of participants. The following section gives some details about tests carried out in order to find suitable values for the parameters.

2.9.5 Testing the selection method

The fitness values in this section are calculated as described in the previous section. Initial tests showed that specifying factors does not increase the results significantly and therefore they are

assumed as one. The decrement value is redundant as for every decrement value the other parameters can be specified accordingly to lead to the same results. Therefore this value is predefined as one too. Calculating the fitness as described has the effect that the tests generated at the beginning tend to have a very high value compared to the other tests and must be prevented from becoming super individuals. Therefore it is again necessary to limit the maximum possible fitness value. As done in previous tests the mutation operation is chosen completely random to compare the values of these tests with the values of Table 14 and Table 15. Table 21 and Table 22 give details of some of the tests carried out in order to get suitable values for the parameters. The tests show that this selection strategy raises the amount of covered lines to 67.46% compared to the 66.57% without this strategy when the picosat solver is tested. The additional amount of 0.89 percentage points corresponds to 21 additional code lines that were executed throughout the tests. The tests with the precosat solver showed that 81.24% of the available code lines were executed. The additional amount of 1.38 percentage points corresponds to 38 additional lines of code. In the case of precosat it will be possible to increase the amount of covered lines by further optimizing parameters. But as there are some drawbacks with this selection strategy as described in the following section this selection strategy was not pursued any further.

Solver		**Picosat**					
Time		**15 min**					
Mutator Selection		**Randomly**					
Input selection		**Tournament with fitness based on gain in coverage**					
Tests							
TournamentSize	**MaxInputFitness**	**Total**	**Sat**	**Unsat**	**TimeOut**	**byte coverage**	**line coverage**
15	10	648	321	327	0	58.91	67.24502124
20	10	665	401	264	0	57.38	66.82363212
10	10	657	368	289	0	58.95	67.16239592
12	10	687	359	328	0	59.02	66.98888275
13	10	640	332	308	0	57.16	66.56749363
12	5	630	321	309	0	56.93	66.52618097
12	15	623	307	316	0	58.4	66.86494477
12	12	605	328	277	0	57.24	66.52618097
12	8	656	301	355	0	57.12	66.60880629
14	10	667	322	345	0	59.33	67.03019541
16	10	428	229	174	25	59.76	67.45984707
18	10	684	393	291	0	59.13	67.37722175

Table 21: Coverage gain based input selection - picosat

Solver		Precosat					
Time		15 min					
Mutator Selection		randomly					
Input selection		Tournament with fitness based on gain in coverage					
Tests							
TournamentSize	MaxInputFitness	Total	Sat	Unsat	TimeOut	byte coverage	line coverage
16	10	484	268	215	1	66.37	81.24264035
18	10	434	236	186	12	64.77	79.71722202

Table 22: Coverage gain based input selection - precosat

2.9.6 Problems of selection from the input cache

Selecting from the input cache as described and tested in the previous sections has some serious drawbacks. First of all it is difficult to find a really good fitness function which handles all the aspects of the tested environment. Inputs which are generated at the beginning of the fuzzing process tend to have a high amount of newly discovered code locations and therefore a high fitness value. Inputs which discover a new input at a later state of the fuzzing process should have a high value because they discovered something that was not discovered before and may be of high interest. Hence the use of the test number as another factor sounded reasonable. The problem is that inputs at a very late state become super individuals immediately if the maximum fitness value is chosen to high. If the maximum fitness value is chosen lower the selection pressure is not as high as expected during the first phases of the fuzzing progress. Although this can be handled by providing a more complex fitness function, e.g. variables of higher order, the overall method does not sound very promising. Adding parameters of higher order will make the selection model even more complicated and it will get unlikely to find suitable and robust values for the parameters which yield the desired effect. Additionally the two strategies defined in the past sections have another problem which is detailed in the following section.

2.9.7 Memory consumption as the general problem of selecting from the input cache

The two selection methods described in the previous sections have a serious problem in common. They depend on saving all inputs which were carried out in order to use them for the mutation step. As detailed in Figure 13 the memory consumption will grow linear with the number of processed inputs. Although this can be handled by limiting the size of the input cache and removing the worst inputs when this limit is exceed the whole selection process becomes very complicated. Additionally it would be hard to define which inputs are the worst because there may not be enough data available for a huge part of the available inputs. If an input which would be very suitable for generating new inputs failed to do so in the first attempt it will be considered as worse input. Therefore it will not be selected in future attempts and has no possibility to increase the fitness value which would make it a candidate for removal from

the input cache. Based on all the problems that were encountered in the previous sections a completely new selection method will be defined and tested in the following sections.

2.9.8 Using basic blocks to select inputs

This selection strategy uses a completely different approach to find a suitable input to be used by the mutators. The idea is based on a work by (Sparks, Embleton, Cunningham, & Zou, 2007) that suggests that code paths in an application which are executed less often should be considered as more interesting. Applied to the information stored in the state of the fuzzing application basic blocks which are tested less often contain code which may be of higher interest. As described in Sec. 2.6.2 every executed basic block stores references to the corresponding inputs. Basic blocks which have a smaller number of referenced inputs can be considered more interesting than others as they demand on certain structure which is unique in the referenced inputs. The technique described in theses sections is implemented by selecting a suitable basic block at first in order to choose an input that covers the selected basic block. A basic block is considered as interesting if little inputs are available which cover the basic block. Again, the selection algorithm uses tournament selection to find basic blocks with little inputs. The tournament size can be specified as argument and gives control over the selection pressure. As depicted in Figure 11 basic blocks may overlap. Additionally basic blocks may have very different lengths which results from the way pin detects and generates them. Therefore basic blocks will not be selected randomly from the available basic blocks but from an address they cover. Therefore prior to selecting a basic block an address is determined whereas all addresses have the same probability to be chosen. Then a basic block which covers this address is used. As addresses are stored in sections which are of different sizes too. Therefore a section is chosen by proportionate selection. The whole selection process is depicted in Figure 18. There are two additional parameters tested which control certain exceptions that may arise if addresses are chosen that have not been executed. As the monitor tool also reports basic blocks and therefore addresses which are detected as part of a trace but have not been executed one parameter specifies whether these basic blacks are allowed to participate in the tournament. If there is no input available at all for a randomly selected address the algorithm searches for nearby addresses which are suitable. This conforms to the assumption that applications are designed according to the locality principle and code at a specific address will be executed more likely if code at surrounding addresses is executed. This strategy implies that addresses which are located near undetected addresses are chosen more often. To limit the advantage of these addresses the search width can be limited by another parameter to the algorithm. If no valid address is found within this limit the algorithm restarts with a new randomly selected address. If the limit is chosen as zero no searches for nearby addresses takes place at all. If a suitable address is found a basic block that contains this address and conforms to the requirements, e.g. the covered address is executed, is chosen randomly. To calculate the fitness of the participants in the tournament the number of available inputs acts as primary

compare operator. If the difference of available inputs is within a certain range, which can be specified in the application configuration, they are considered as equal and the generation when the basic block was executed the first time is used to compare the basic blocks.

The idea is that basic blocks which were detected later in the testing phase may be of more interest. If both basic blocks were executed at the same Generation, or none of the two has been executed at all which is more likely, the Generation when the basic block was discovered by pin is used to compare them. Finally an input that executes or recognizes this basic block is chosen via a tournament selection strategy. In this tournament inputs that execute the basic block more often are preferred to others. The idea behind this strategy is that executing a basic block more often will execute the surrounding loop more often and therefore the whole strategy targets towards the limits of loops.

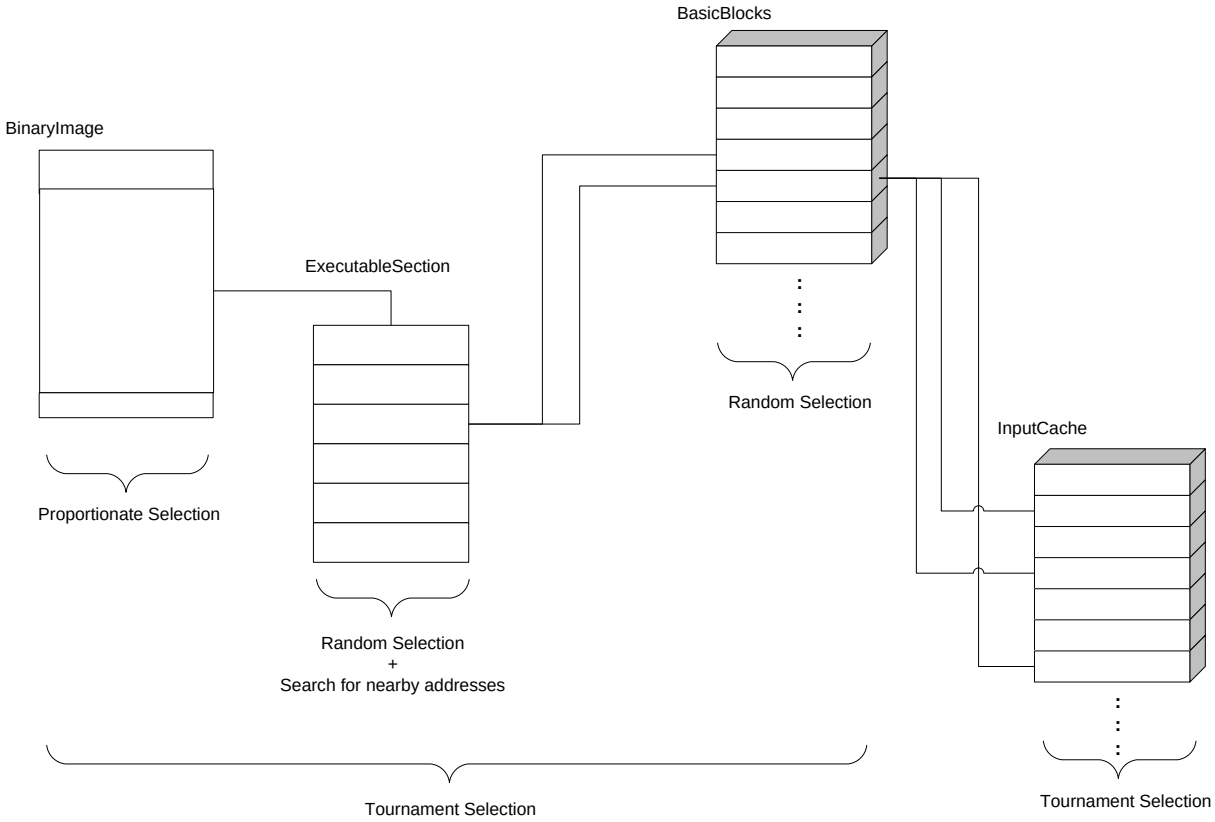

Figure 18: Selecting inputs via basic blocks

2.9.9 The idea behind selecting from basic blocks

This section describes a hypothetical example which highlights the idea behind the selection strategy described in the previous section. Figure 19 shows the layout of an assumed application with 5 basic blocks.

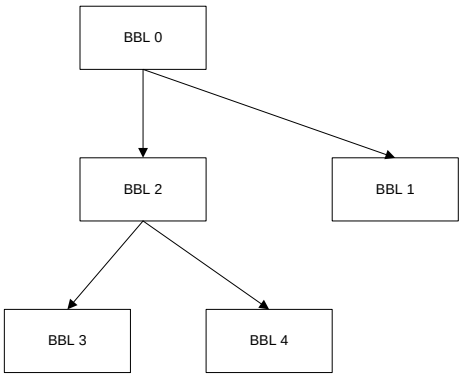

Figure 19: BBL selection static basic blocks

Figure 20 shows the application state assuming two inputs were generated and processed which executed only the basic blocks BBL 0 and BBL 1. Therefore the number of available inputs at these basic blocks is two.

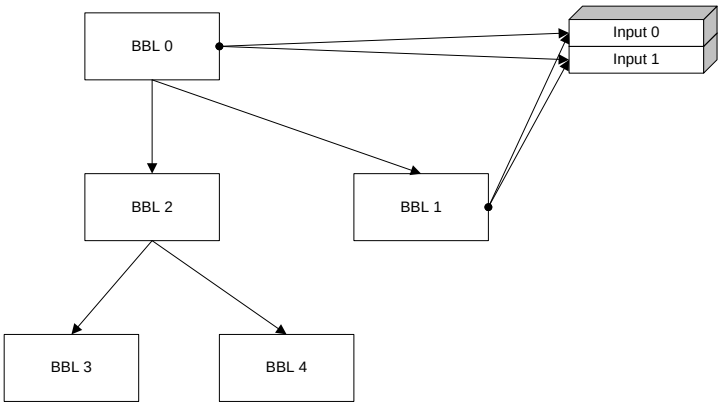

Figure 20: BBL selection after two inputs

Figure 21 shows the application state after processing a further input that is assumed to execute the basic blocks BBL 0, BBL 2 and BBL 3. The available inputs at these basic blocks are now one and lower than the number of available inputs at other basic blocks which were executed. Therefore these two basic blocks are preferred by the basic block selection process. If any of the two basic blocks will be selected the input which is used by the mutation step will be Input 3. Now the chances are high that the input will be modified by the used mutator to execute BBL 4, e.g. by executing BBL 0, BBL 2 and BBL 4 as shown in Figure 22. In this case BBL 3 and BBL 4 will be preferred in further basic block selection operations as they have the least number of available inputs. If BBL 4 were not executed by Input 4, Input 3 would still have a

higher probability to be selected as the number of available inputs at BBL 2 and BBL 3 would be lower than the amount of available inputs at other basic blocks and chances are again high that BBL 4 will be executed by further inputs.

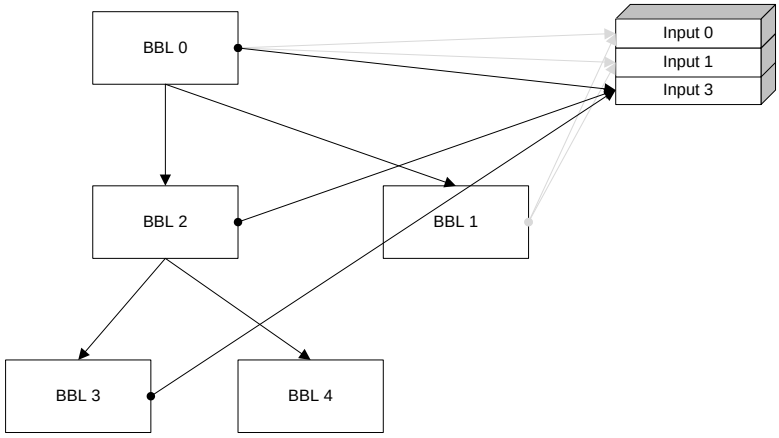

Figure 21: BBL selection after three inputs

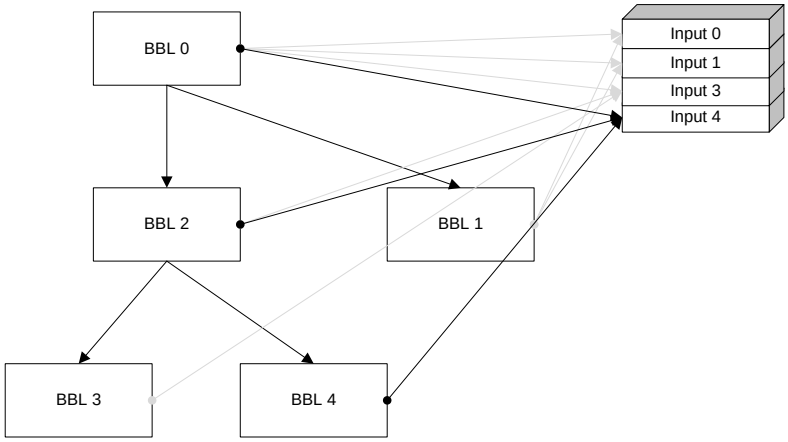

Figure 22: BBL selection after four inputs

2.9.10 Testing the strategy of choosing inputs from suitable basic blocks

Table 23 and Table 24 show details about a number of tests that were carried out in order to get suitable values experimentally for the parameters of the input selection strategy described in the previous sections. Initial tests uncovered that allowing basic blocks to participate in the tournament, which have not been executed yields better results. Therefore all the tests listed in the tables allowed them to participate. Using this selection strategy raised the amount of covered lines to 67.67% compared to 66.57% when inputs were chosen randomly in case the picosat solver was tested. The additional amount of 1.10 percentage points corresponds to 26 additional code lines that were executed throughout the tests. The tests with the precosat solver showed that 81.31% of the available code lines were executed. The additional amount of

1.45 percentage points corresponds to 40 additional lines of code which were executed during the tests.

Solver	Picosat								
Time	15 min								
Mutator selection	Randomly								
Input selection	Based on suitable basic blocks								
Tests									
TournamentSize	Input Tournament	SearchOffset	MinInputsDiff	Total	Sat	Unsat	TimeOut	byte coverage	line coverage
9	5	10	1	509	299	202	8	59.77	67.4598471
9	10	10	1	555	308	247	0	59.52	67.4598471
9	8	10	1	611	360	251	0	59.34	67.2450212
9	20	10	1	512	301	210	1	59.88	67.6746729
9	30	10	1	585	335	250	0	58.65	67.0301954
9	15	10	1	606	386	220	0	58.89	67.0301954
9	20	50	1	584	397	183	4	59.67	67.4598471
9	20	20	1	575	389	185	1	59.54	67.4598471
9	20	5	1	533	292	237	4	59.69	67.4598471
15	20	10	1	590	350	240	0	59.2	67.0301954
9	20	10	2	411	248	161	2	59.61	67.4598471

Table 23: Basic block based input selection - picosat

Solver	Precosat								
Time	15 min								
Mutator selection	Randomly								
Input selection	Based on suitable basic blocks								
Tests									
TournamentSize	Input Tournament	SearchOffset	MinInputsDiff	Total	Sat	Unsat	TimeOut	byte coverage	line coverage
9	20	10	1	408	223	183	2	65.4	80.7919486
13	20	10	1	463	255	208	0	66.06	80.7919486
9	20	20	1	474	278	196	0	64.88	80.4019268
9	30	10	1	430	238	191	1	65.03	79.9685694
9	20	10	2	497	296	200	1	66.36	81.3119775
14	20	10	3	470	278	192	0	64.58	79.717222
14	20	10	2	357	192	151	14	66.36	81.1993046

Table 24: Basic block based input selection - precosat

2.9.11 Problem if cached input is not processed

As described in Sec. 2.6.3 an input cache is used to prevent the generation of traces for the same input multiple times. Besides the problems described in Sec. 2.6.2 there may be an additional problem if caching of inputs is enabled which is unique to the selection strategy described in these sections. It is possible that inputs with a particular characteristic exist which

execute a certain basic block and it is not possible to create different inputs with the same behavior. For instance there is only one possible input which contains no variables. If this input executes basic blocks which were not covered by other inputs they will become super individuals and will be chosen over and over again. As the input is unique it is not possible to generate further inputs that execute these basic blocks. The problem gets even worse when more and more inputs will be available at other basic blocks during the fuzzing progress as therefore the fitness of the basic blocks that refer to the unique inputs increases indirectly. There are two possibilities to overcome with this problem. The first and easiest one is to disable the execution cache altogether. This has the drawback that equal tests may be run several times which causes performance penalties. The alternative strategy is to include the trace information of a test run into the input cache. With this information it is possible to update all the data in the execution cache as if the actual test was run. The drawback is that the consumption of memory will rise accordingly as traces tend to be very large. Sec. 2.12 describes the benefits and drawbacks of using the input cache in general and as it comes to realize that the overall advantage of using the input cache is very limited the final version of the fuzzing application does not have one.

2.9.12 Conclusion of the input selection methods

Table 25 and Figure 23 give details about all the tested input selection strategies. The amount of covered code lines is very similar for each of the three tested strategies. Selection from the input cache using fitness increments may be preferred to selection from the input cache based on the gain in coverage as it leads to very similar results but is simpler and more robust.

Time	**15 min**			
Mutator selection	**randomly**			
	picosat		**precosat**	
input selection method	**line coverage**	**Num tests**	**Line coverage**	**Num tests**
inputcache with fitenss increment	67.29	604	81.31	461
inputcache based on coverage gain	67.45	428	81.24	484
basic block selection	67.67	512	81.31	497

Table 25: Conclusions of the input selection methods

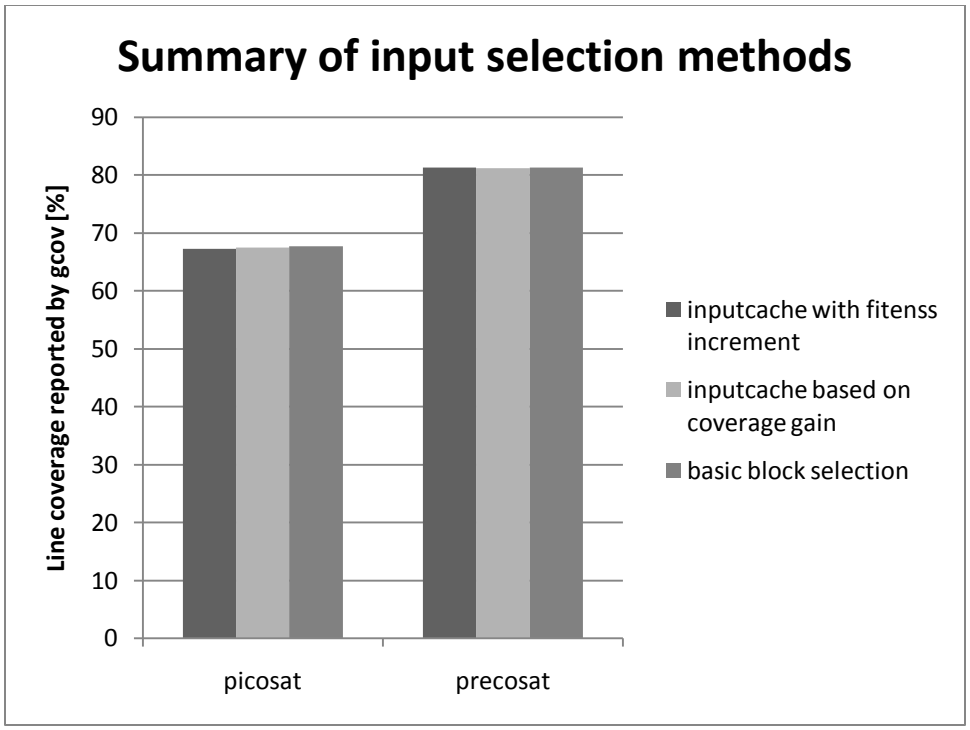

Figure 23: Summary of input selection methods

2.9.13 Selecting the input selection method

This section describes a final modification to the input selection method. Some kind of meta-selection method is implemented to select between two input selection strategies which are the selection from the input cache using fitness increments and selection of inputs from suitable basic blocks. This is implemented by using a fitness value for each selection method. The fitness value is increased if the selected input yields a successful result. The value is decremented every time the corresponding selection method is used whereas the value will not be decremented below one. Proportionate selection is used to choose the selection method. This method does not yield any additional covered code but it was possible to discover an interesting phenomenon. In the initial phases of the fuzzing process inputs from the input cache were preferred. That way the fuzzing algorithm succeeded in covering a great amount of code within a short time. In later phases of the fuzzing progress the method of selecting inputs from suitable basic blocks was preferred. Based on this observation it is concluded that selecting an input from interesting basic blocks will execute code which requires special input more likely.

2.9.14 Check which BBLs and or Inputs are chosen

To get information about basic blocks and inputs which were chosen during the fuzzing progress some information about the most used and the most successful used basic blocks and inputs are reported by the fuzzing application. Some of them are listed in the following tables. Table 26 shows that inputs which were used often also yielded successful results. Table 27 shows that this is not the case generally when basic blocks are selected. The reason for this is

that due to the huge number of basic blocks only one of them was chosen a second time. Therefore the testing time, which was 15 minutes, was far too short to give useful results. This issue will be addresses when the fuzzing application is modified to support testing for larger durations which is not the case currently due to the linear growing of memory consumption.

Solver	**picosat**	
Time	**15 min**	
Mutation selection	**randomly**	
Input selection	**meta-based selection**	
Inputs sorted by #Used		
Input	**#Used**	**#Succ**
p cnf 126 2197	8	2
p cnf 53 218	8	3
p cnf 212 2524	7	1
p cnf 61 674	7	2
p cnf 18 225	7	3
p cnf 131 2455	6	1
p cnf 136 2524	6	1
p cnf 61 674	4	1
p cnf 101 1860	4	1
p cnf 120 1680	4	2

Table 26: Input statistics

Solver	**picosat**		
Time	**15 min**		
Mutation selection	**randomly**		
Input selection	**meta-based selection**		
Addresses sorted by #Used			
Address	**#Used**	**#Succ**	**#Inputs**
66529	2	1	455
101585	1	0	13
98163	1	0	11
98007	1	1	40
97752	1	0	62
96373	1	0	56
95873	1	0	4
95604	1	0	8
95296	1	0	54
95248	1	0	8

Table 27: Basic block statistics

2.9.15 Conclusion of using feedback to select input

Details of the amount of covered code lines are given in Table 28 and Figure 24 whereas line coverage with the current version of the fuzzing tool is compared to using the cnfuzz tool. With the strategies described in the previous sections it was possible to nearly reach the amount of covered code lines when fuzzing the picosat solver. When the precosat solver was fuzzed the line coverage could be increased by 3.2 percentage points which corresponds to 88 additional code lines which were executed during the test which is a notable amount. A reason why the coverage of the picosat solver could not be increased significantly is that the code reachable through the given input vector has been covered already very well by the cnfuzz tool. The entire tests carried out in this and the previous sections are of little validity as a testing time of 15 minutes is far too short to give valid results as already mentioned. Before longer durations can be tested another problem must be addressed in the following sections. The problem with the current implementation is that memory consumption still grows linear with the amount of tests carried out which causes the performance of the fuzzing application to drastically drop after about one hour of testing time.

Time	**15 min**			
	picosat		**precosat**	
fuzzer	**line coverage**	**#tests**	**line coverage**	**#tests**
cnfuzz	67.72	9033	78.11	7830
fuzz with input selection	67.67	512	81.31	497

Table 28: Coverage with input selection

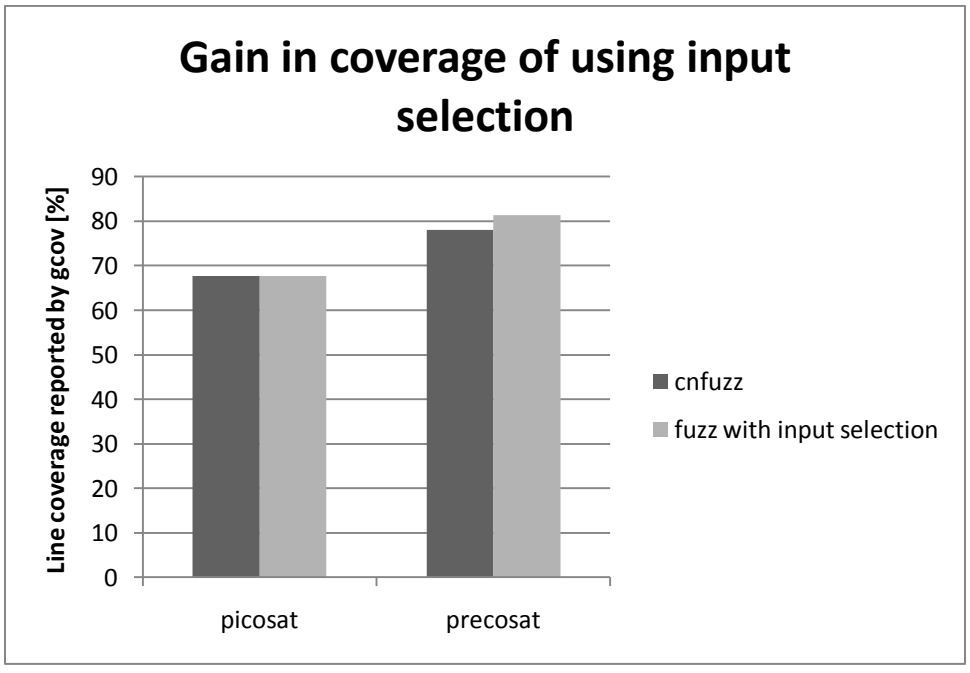

Figure 24: Gain in coverage of using input selection

2.10 Effect of checking for duplicates

The main reason for using an input cache was to prevent the algorithm from generating traces for equal inputs multiple times. This is based on the fact that the generation of the application trace is by far the most time consuming step in the fuzzing progress as described in Sec. 2.5. The drawback of using an input cache is that the consumed memory grows linearly with the number of tests carried out. As described in more detail in Sec. 2.9.11 another problem which is unique to selecting suitable basic blocks to choose an input occurs if certain inputs are omitted as the statistics about the basic blocks are not updated. As already mentioned there are two possible solutions to solve the memory problem. The first one is to limit the size of the input cache and to remove inputs from the cache if the limit is exceeded. The second one is to completely remove the input cache and accept the additional calculation time which is used to calculate inputs otherwise found in the cache. The following section gives some details about the benefit of the input cache and as the result is that the input cache is of little use it will be removed in the final version of the fuzzing application.

2.11 Checking the effects of using an input cache

This section estimates the benefit of using the input cache by running the fuzzing application with the SAT solver picosat for certain amounts of times as shown in Table 29. For each test the number of inputs tested and the number of inputs omitted by the input cache is given. These tests make it clear that the amount of inputs found in the cache is far too low to make its use beneficial. The table also shows that the number of tests performed within the given amount of time decreases as the time for searching the input cache grows and the overall system load increases. If the last test is considered in detail only 3.9% of the inputs were found in the cache. If the cache is omitted it would have taken an estimated amount of 2 minutes to also test these 62 inputs.

testing the input cache		
solver	**Picosat**	
Tests		
time	**#tests**	**#found in cache**
10	435	5
30	902	43
60	1606	62

Table 29: Effect of using the input cache

2.12 Conclusions of testing the benefit of the input cache

As described in the previous section the input cache is of little use and is removed in the final version of the fuzzing application. This implies that several input selection methods as described in Sec. 2.9 become impossible and the method of selecting inputs from suitable basic

blocks remains the only available selection method. Therefore the final version of the fuzzing application will use this input selection method.

2.13 Resolving the memory problem

Removing the input cache from the fuzzing application, as described in the previous sections does not resolve the problem of consumed memory as every executed basic blocks still stores a reference to the inputs that cover it. Tests carried out in Sec. 2.9.10 show that the input selection strategy gives better results if the selection pressure to choose a suitable input from a selected basic block is high. Therefore some of the information stored in the fuzzing application described in Sec. 2.6.2 will be removed and basic blocks will solely store a reference to the most suited basic block. This limits the amount of overall stored inputs as the amount of statically available basic blocks is limited by the target application size. Additionally the size of inputs stored at these basic blocks is limited by the amount of calculation time given to the target application. Figure 25 shows the amount of consumed virtual memory after the fuzzing application has been updated to solely store the most suited input. As the diagram shows the amount of consumed memory is far below the amount of memory necessary when previous methods were implemented. The amount still rises as more and more basic blocks are executed during the fuzzing process and will be stored in the application state. Additionally the inputs stored at the basic blocks tend to get larger as the fuzzing application tries executing basic blocks most often which is usually given by larger inputs. Nevertheless there will be an upper limit for the total amount of memory consumed as described previously in this section.

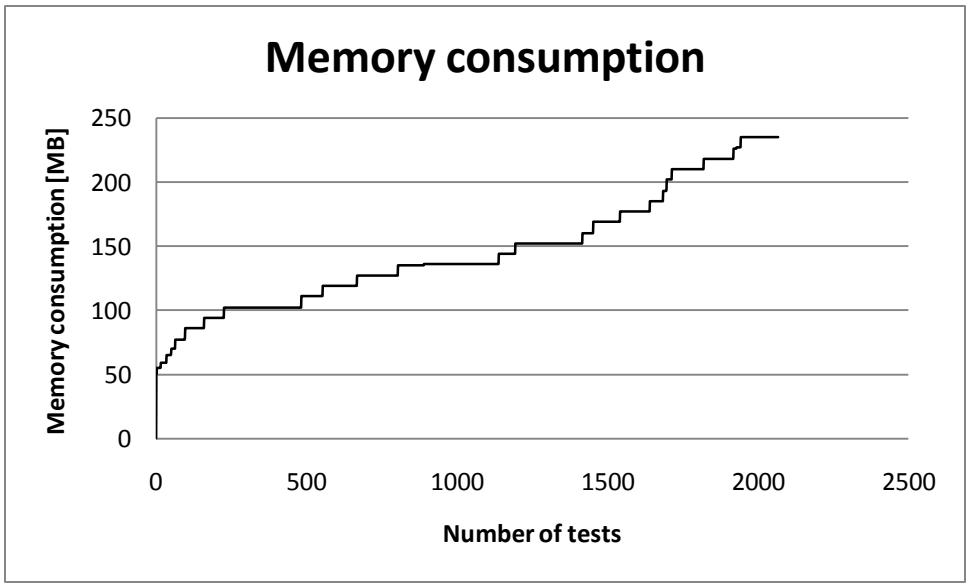

Figure 25: Memory consumption without storing all inputs

2.13.1 Adapting the basic block selection method to the solved memory problem

After the input cache has been removed from the fuzzing application and basic blocks store the most suited input solely it is necessary to adjust certain behaviors of the way suitable mutators and basic blocks are selected. To start with the tournament selection step to get a suitable input from a selected basic block has become obsolete as only the most suited input is stored. As this input can be updated several times during the fuzzing process an additional piece of information is available, which is the latest time that happened. This information is used to additionally update the fitness of mutators if they succeeded in generating an input that caused an update. Updates of the most suited inputs are performed every time an input is found which executes the basic block more often or a shorter input is found which executes the basic block equally often. Finally checking if a basic block is more suited than others is adapted to consider the time when basic blocks were last updated in case the difference of the number of available inputs is within a specified range as described in Sec. 2.9.8. Table 30 gives some details about tests carried out with the updated version of the fuzzing application in order to get suitable values for the parameters to increase the fitness in case basic blocks were updated experimentally.

Solver		**picosat**					
Time		**15 min**					
Mutator selection		**Proportionate selection**					
Tests							
Bonus	**UpdateBBLBonus**	**Total**	**Sat**	**Unsat**	**TimeOut**	**byte coverage**	**line coverage**
15	1	613	387	226	0	59.75	67.7182073
10	1	612	405	207	0	59.75	67.7182073
10	2	557	475	81	1	59.85	67.4598471
5	1	602	353	217	0	59.7	67.157689
8	1	602	312	290	0	59.49	67.2450212

Table 30: Optimizing basic block update bonus

2.14 Final adaption of parameters

As already mentioned there are correlations between the used parameters and techniques. Therefore a final test was carried out to optimize some of the untested parameters using the picosat solver and a testing time of 30 minutes. Some of the values are given in Table 31 and the values with the highest amount of covered code lines are used in the final version of the fuzzing application.

Solver	picosat										
Time	30 min										
Mutators	All										
Tests											
Total	Sat	Unsat	TimeOut	Undef	byte coverage	line coverage	BBLTournamentSize	UseBBLFitness	BBLFitnessBonus	SelBBLMaxSearchOffset	SelBBLMinNumInputsDiff
1086	683	389	8	0	60.1	67.8457009	15	False	3	20	1
1056	651	404	1	0	59.95	67.7159856	30	False	3	20	1
1252	720	532	0	0	59.8	67.2450212	20	False	3	20	1
1140	700	439	1	0	59.75	67.5033815	30	True	3	20	1
1145	703	441	1	0	59.75	67.5033815	30	True	5	20	1
1240	698	542	0	0	59.69	67.0797706	30	False	3	30	1
1197	750	447	0	0	59.57	67.2863339	30	False	3	10	1
1168	603	563	2	0	59.94	67.6746729	30	False	3	20	2

Table 31: Further optimization of parameters

2.15 Final results of the tests with valid inputs

Table 32 shows some information of running the solvers picosat and precosat with the current version of the fuzzing application and the specified configuration for one hour. The inputs generated by this test are all valid as it is the case when cnfuzz was used and therefore the amount of covered code lines can be compared between the two fuzzers. This is done in the next section.

time	60 min						
Tests							
solver	#tests	#sat	#unsat	#timeout	#undefined	byte coverage	linecoverage
picosat	1913	1116	753	36	0	60.46	67.93
precosat	1806	1042	687	36	41	67.56	82.40

Table 32: Testing the solver applications for one hour with valid formulas

2.16 Comparison of the current strategy with cnfuzz

In this section the results of using the current implementation of the fuzzing application are compared with the results of the cnfuzz tool. Table 33 shows a summary of the amount of covered code lines and Figure 26 depicts the results. As the table shows the amount of covered lines could be increased by an amount of 1.29 percentage points if the precosat solver was fuzzed. However the amount of covered code lines when fuzzing the picosat solver is slightly lower than the amount covered by the cnfuzz tool. Therefore some detailed analyses of the differences in the code coverage concerning the picosat solver are given in the next section. Nevertheless from the information in the tables it can be seen that the test inputs generated are of a very high quality. It was only necessary to carry out about $1/20^{th}$ of the tests to achieve or top the amount of code lines covered by the cnfuzz tool.

time	60 min			
	cnfuzz		fuzz valid formulas only	
solver	#tests	line coverage	#tests	line coverage
picosat	43138	67.97	1913	67.93
precosat	36412	81.11	1806	82.40

Table 33: Results of fuzzing valid formulas

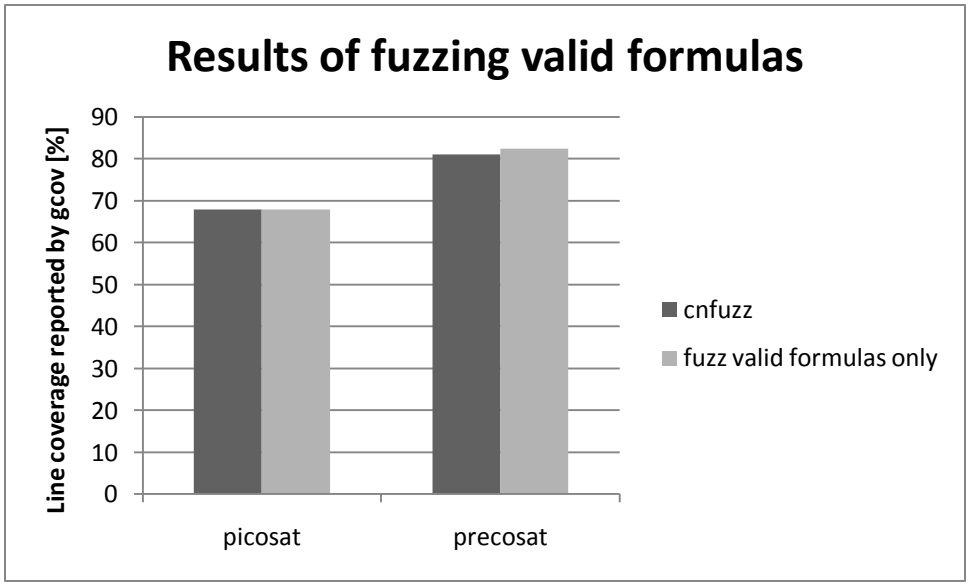

Figure 26: Results of fuzzing valid formulas

2.16.1 Analyses of the differences in code coverage

In this section the reasons for not covering certain code lines when fuzzing the picosat solver are analyzed by comparing information gained by the gcov utility. Table 34 shows the number of uniquely hit code lines for each fuzzer and the corresponding source files. Table 35 gives details of the code lines which were executed during the tests with cnfuzz but not with the fuzzing application. As can be seen in the table the six code lines which were not executed by the fuzzing application are within two code blocks. The condition guarding the first code block was executed 222 times by cnfuzz but the corresponding else path was executed only one time. It is also worth to note that a total number of 43138 tests were necessary to execute that block one time. It is reasonable that the code was not executed by the fuzz application as the condition which guards the else block was only executed four times which limits the probability to enter the else path. The second block of code considered was executed two times by the cnfuzz application whereas it was necessary to execute the guarding condition 36008 times. The fuzzing application executed the guarding condition 1992 times but did not succeed in entering the block. If the number of executions of the condition is considered closer it can be assumed that a mean number of 18004 tests are required to enter the code block one time. The cnfuzz application carried out 22.55 times more tests than the fuzz application. If this factor is

multiplied by the number of times the condition was executed by the fuzz application a hypothetical result of 44919 executions could be expected. This means that the selection algorithm implemented by the fuzzing application preferred this address which shows the benefits of the selection and mutation steps. This reasoning is not true for the first code block which can be explained by the fact that the overall number of executions of the guarding condition is very low for both fuzzers. Probably the fuzzing application succeeded very late in the fuzzing progress to cover the guarding condition and had not enough tests left until the additional pressure on the condition got effective.

SourceFile	Solely hit by fuzz valid formulas only	Solely hit by cnfuzz
main.c	0	0
app.c	4	0
picosat.c	1	6
Total	**5**	**6**

Table 34: Uniquely hit code lines

cnfuzz		fuzz valid formulas only	
Location	**Code**	**Location**	**Code**
222: 5130:	if (conflict)	4: 5130:	if (conflict)
-: 5131:	{	-: 5131:	{
221: 5132:	backtrack ();	4: 5132:	backtrack ();
-: 5133:	assert (!level);	-: 5133:	assert (!level);
-: 5134:	}	-: 5134:	}
-: 5135:	else	-: 5135:	else
-: 5136:	{	-: 5136:	{
1: 5137:	assign_decision (NOTLIT (pivot));	#####: 5137:	assign_decision (NOTLIT (pivot));
-: 5138:	flbcp ();	-: 5138:	flbcp ();
1: 5139:	backtrack ();	#####: 5139:	backtrack ();
-: 5140:		-: 5140:	
1: 5141:	if (level)	#####: 5141:	if (level)
-: 5142:	{	-: 5142:	{
-: 5143:	assert (level == 1);	-: 5143:	assert (level == 1);
-: 5144:	flbcp ();	-: 5144:	flbcp ();
-: 5145:		-: 5145:	
1: 5146:	if (!conflict)	#####: 5146:	if (!conflict)
-: 5147:		-: 5147:	
-: 5148:	#ifdef STATS	-: 5148:	#ifdef STATS
-: 5149:	floopsed++;	-: 5149:	floopsed++;
-: 5150:	#endif	-: 5150:	#endif
#####: 5151:	undo (0);	#####: 5151:	undo (0);
-: 5152:	continue;	-: 5152:	continue;
-: 5153:	}	-: 5153:	}
-: 5154:		-: 5154:	
1: 5155:	backtrack ();	#####: 5155:	backtrack ();
-: 5156:	}	-: 5156:	}
36008: 5259:	if (delta > 2000000)	1992: 5259:	if (delta > 2000000)
2: 5260:	delta = 2000000;	#####: 5260:	delta = 2000000;

Table 35: Analysis of code that was not hit by fuzz

2.17 Further increasing the coverage

As already mentioned in Sec. 1.6.1 there are two main reasons for not covering a great amount of code during the fuzzing process which are invalid inputs and arguments to the solving application. These two issues will be addresses in the following sections.

2.17.1 Generate invalid input

Many parts of the SAT solver's code will be executed only if invalid input is provided. All the tests described in the past sections generated valid inputs in terms of the DIMACS format to compare the results with the results of running the cnfuzz tool. The following sections address

the issues of generating invalid inputs by adding some additional mutators which operate on the stream that is sent to the targeted SAT solving application on a character basis.

2.17.2 The baseline test

To test the influence of fuzzing the input stream another baseline test is carried out which runs the SAT solver picosat for 15 minutes. In this test all mutators that operate on the model of formulas are used and details of the test run are given in Table 36.

solver	**picosat**					
time	**15 min**					
Mutators	**CNF Model only**					
Test						
Total	**Sat**	**Unsat**	**TimeOut**	**Undef**	**byte coverage**	**line coverage**
497	260	231	6	0	59.97	67.45984707

Table 36: Generation of valid formulas with the fuzzing application

2.17.3 ModifyChar

This mutator changes the characters of the stream sent to the solving application. The number of characters which are modified is chosen from an exponential distribution which is given as a parameter to the mutator. The characters which are modified are chosen equally distributed from all the characters in the stream. The new value of a character is chosen from a Gaussian distribution with the old value of the character as mean. The deviation of the Gaussian distribution is specified by another parameter to the mutator. Table 37 shows some tests that were carried out in order to find suitable values for the parameters to the mutator experimentally.

Solver	**picosat**							
Time	**15 min**							
Mutators	**CNF Model + ModifyChar**							
ModifyChar Mean	**Modify Char Deviation**	**Total**	**Sat**	**Unsat**	**TimeOut**	**Undef**	**byte coverage**	**line coverage**
20	5	631	309	263	0	8	59.32	67.33475786
5	3	627	346	242	0	8	59.12	66.82181393
100	3	663	410	182	0	8	58.92	67.28518267
50	3	360	165	136	32	10	59.81	67.71483432
50	5	612	345	209	0	2	59.57	67.71483432

Table 37: Optimize ModifyChar

2.17.4 DuplicateChar / DeleteChar

These mutators duplicate or delete characters in the input stream sent to the solving application. The number of characters which are added or deleted is again chosen from an exponential distribution which is given as a parameter to the mutator. The position where

characters are added or deleted is chosen equally distributed from all the available positions in the stream. Table 38 shows some tests that were carried out in order to find suitable values for the parameters.

Solver	**picosat**							
Time	**15 min**							
Mutators	**CNF Model + DuplicateChar + DeleteChar**							
DuplicateChar mean	**DeleteChar mean**	**Total**	**Sat**	**Unsat**	**TimeOut**	**Undef**	**byte coverage**	**line coverage**
5	5	680	291	285	0	7	59.51	67.37069669
20	20	644	319	217	0	11	59.46	67.84232795
50	50	620	315	183	0	24	60.05	67.80034834
35	35	698	323	254	0	25	59.2	67.62750212

Table 38: Optimizing DuplicatChar and DeleteChar

2.17.5 AddWellKnownPhrase

This mutator is specialized on the DIMACS format used to serialize problems in CNF. The mutator adds phrases at uniquely distributed positions in the stream. The number of add operations is chosen exponentially distributed. The phrases which are added are chosen uniquely distributed from a set of strings which are known to occur in formulas serialized in DIMACS format. Table 39 shows the phrases which are used in the implementation of the fuzzing application and Table 40 shows some tests that were carried out in order to find suitable values for the parameters.

c
p
cnf
0
1
-
-0
-1

Table 39: Well known phrases

Solver	**picosat**						
Time	**15 min**						
Mutators	**CNF Model + AddWellKnownPhrase**						
Tests							
Mean	**Total**	**Sat**	**Unsat**	**TimeOut**	**Undef**	**byte coverage**	**line coverage**
5	649	339	247	0	1	59.57	67.62750212
20	619	309	249	0	0	59.68	67.4126763
10	414	210	131	26	4	60.13	67.80034834

Table 40: Optimizing AddWellKnownPhrase

2.17.6 Test with all stream mutators enabled

This section shows the amount of code that was covered if all the mutators described in the previous sections were enabled. Details are given in Table 41 and Figure 27. As can be seen in the diagram the overall amount of covered code is slightly below the amount of coverage if only valid formulas were generated. More detailed analysis showed that additional code which parses the formulas was covered hence fewer tests were carried out which targeted the other parts of the solving application. The fact that the amount of code which parses the input is rather small explains why the overall amount of coverage decreased.

time	60 min						
Tests							
solver	#tests	#sat	#unsat	#timeout	#undefined	byte coverage	linecoverage
Picosat	2545	1296	691	1	0	60.87	67.93
precosat	2132	1253	569	3	0	67.09	81.93

Table 41: fuzz with stream mutators

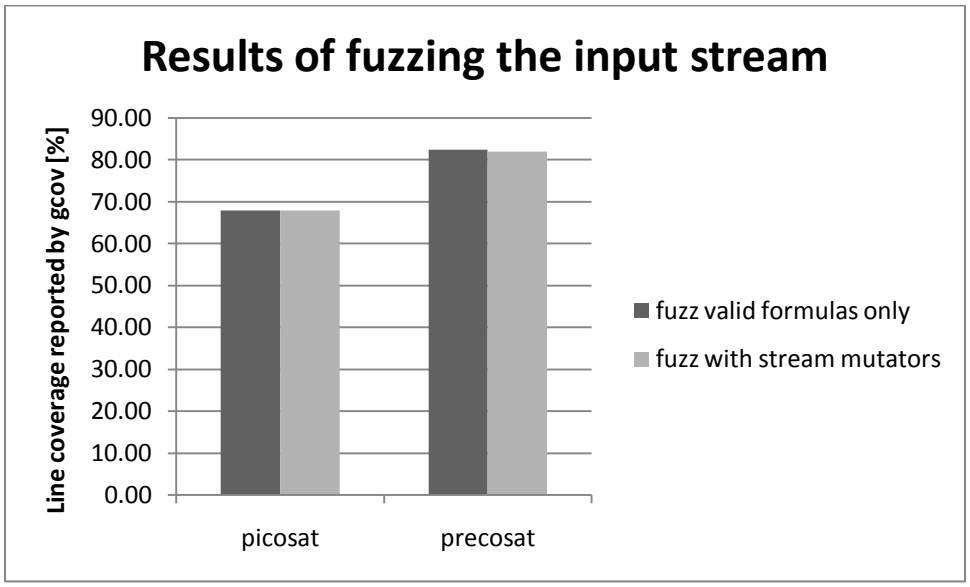

Figure 27: fuzz with stream mutators

2.17.7 Fuzz arguments

These sections describe several methods that were implemented in order to fuzz arguments to the target application. A list of available arguments can be provided to the fuzzing application as a file containing a line feed separated list of them. Figure 28 shows exemplarily the arguments file used to fuzz the picosat solver. This file is processed by the fuzzing application in order to generate a list of predefined arguments which helps the fuzzer to create valid arguments more easily. If no arguments are specified the fuzzer tries to generate random strings which is very unlikely to form valid arguments.

```
-h
--version
--config
-v
-f
-n
-p
-a
-a 0
-a 1
-a -1
-l 100
-i 0
-i 1
-s 123
-o out.file
-t trace.file
-T Trace.file
-r rev.file
-R rup.file
-c core.file
-V Core.file
-U coreU.file
-
test.gz
```

Figure 28: Arguments file of picosat

2.17.8 AddWellKnownArgument

This mutator chooses one of the available arguments as listed in Figure 28 and adds it to the list of used arguments stored for every input whereas the argument to be added is selected uniquely distributed. There are no additional parameters required for this mutator. Table 42 gives details of a test carried out in order to check the amount of additionally covered code lines if this operator is used. As can be seen in the table the amount of executed lines could be raised by 15.51 percentage points up to 82.97%. This corresponds to 365 lines of code that were additionally executed. This test also shows that fuzzing arguments which are passed to the solver application is very important in order to maximize overall coverage.

Solver	**picosat**					
Time	**15 min**					
Mutators	**CNF Model + AddWellKnownArgument**					
Tests						
Total	**Sat**	**Unsat**	**TimeOut**	**Undef**	**byte coverage**	**line coverage**
725	347	150	0	68	74.26	82.9674894

Table 42: AddWellKnownArgument

2.17.9 Search for inputs that require arguments

Before additional mutators which operate on the arguments passed to the solver application are introduced in the next sections another issue specific to working with arguments needs to be addressed. All of the mutators described in the next sections require inputs which have

arguments e.g. removal of an argument makes no sense if no argument is available. To search for inputs with arguments the input selection strategy, described in Sec. 2.9.8, can be instructed to accept only inputs that contain arguments. This is implemented by restarting the basic block selection process if the basic block selected contains no input with arguments. To prevent the search algorithm to continue searching forever in case no input with arguments is available the search process abandons if no inputs are found within a certain amount of tests. If this is the case the mutator used will be set to AddWellKnownArgument and new arguments will be added to the inputs in the application state. The following sections describe mutators that are used if a suitable input with arguments has been found.

2.17.10 DeleteArgument

This mutator removes any of the available arguments of a selected input. The number of arguments which will be removed by the mutator is chosen from an exponential distribution. This mutator is not primarily expected to uncover new code but is necessary as the fuzzing application will continuously add arguments otherwise. As already mentioned in Sec. 2.13.1 the application state stores a reference to the most suited input for each basic block. The determination of the most suited input has been updated to prefer the simpler input in case there are two inputs that executed the basic block equally often. This means that inputs with fewer arguments are preferred in order to limit the amount of arguments passed to the target application. Table 43 gives some details of tests carried out in order to find a suitable value for the number of arguments that should be removed by the mutator experimentally.

Solver	**Picosat**						
Time	**15 min**						
Mutators	**CNF Model + AddWellKnownPhrase + DeleteArgument**						
Tests							
DeleteArgumentMean	**Total**	**Sat**	**Unsat**	**TimeOut**	**Undef**	**byte coverage**	**line coverage**
1	685	352	171	0	42	74.91	82.7533305
2	674	298	208	0	44	74.67	82.54321155
0.5	711	335	196	0	59	73.58	82.27966015

Table 43: Optimizing DeleteArgument

2.17.11 DeleteArgumentChar

This mutator removes characters from the arguments stream. The number of characters to delete is chosen from an exponential distribution and is given as parameter to the mutator. The main purpose of this mutator is to generate invalid arguments. Table 44 gives some details of tests carried out in order to get a suitable value for the number of characters which should be deleted experimentally.

Solver	Picosat						
Time	15 min						
Mutators	CNF Model + AddWellKnownPhrase + DeleteArgumentChar						
Tests							
DeleteCharMean	Total	Sat	Unsat	TimeOut	Undef	byte coverage	line coverage
1	427	126	75	30	56	76.23	83.8104503
2	443	163	86	15	28	75.87	83.68517842
0.5	458	174	84	24		22	83.26156754

Table 44: Optimizing DeleteArgumentChar

2.17.12 DuplicateArgumentChar / ModifyArgumentChar

This mutator DuplicateChar duplicates a randomly chosen character in the arguments stream. The number of characters which should be duplicated is chosen from an exponential distribution. The purpose of this mutator is to generate additional characters in the arguments stream which could be used by the mutator ModifyChar. This mutator uses a number of randomly chosen characters in the arguments stream and modifies their value. The number of characters is chosen from an exponential distribution. The new value of the character is chosen from a Gaussian distribution with the old value of the character as mean. The deviation of the Gaussian distribution is passed as argument to the mutator. The purpose of these mutators is to generate arguments which are not specified in the arguments file. As already mentioned in the introduction the fuzzing application uses black block techniques solely. The arguments file can be generated by getting available arguments from the usage information or the documentation of the target application. Nevertheless often not all possible arguments are described in these documents, as it was the case with the tested SAT solvers. By using these two mutators it will also be possible to generate any undocumented arguments randomly. A number of tests in order to find a suitable value for the arguments to the mutators are detailed in Table 45.

Solver	picosat								
Time	15 min								
Mutators	CNF Model + AddWellKnownPhrase + DuplicateArgumentchar + ModifyArgumentChar								
Tests									
DuplicateCharMean	ModifyCharMean	ModifyCharDeviation	Total	Sat	Unsat	TimeOut	Undef	byte coverage	line coverage
1	1	3	747	326	131	0	56	72.92	82.45183942
1	2	5	712	252	138	0	74	74.76	83.09383178
1	3	5	676	330	109	0	72	75.84	83.3511215
1	6	5	749	247	159	0	92	73.62	82.79771453
1	3	10	717	313	122	0	78	75.44	82.98890399
3	3	5	732	329	132	0	84	75.34	83.07441801

Table 45: Optimizing DuplicateArgumentChar and ModifyArgumentChar

2.17.13 Tests with all argument mutators

In this test all mutators which were described in this thesis were enabled. Again both SAT solvers were fuzzed for one hour. Details about the amount of coverage are given in Table 46 and Figure 29. These tests showed that the overall line coverage could be increased to 84.71% when the picosat solver was tested and to 87.72% when the precosat solver was tested. Compared to the tests in Sec. 1.6, the line coverage could be increased by 16.74 percentage points which corresponds to 393 code lines in the case of picosat. The tests using the precosat solver showed that the line coverage could be increased by 6.61 percentage points, which is equal to 182 lines of additionally executed code.

time	60 min						
Tests							
solver	#tests	#sat	#unsat	#timeout	#undefined	byte coverage	linecoverage
picosat	2816	972	278	0	339	76.41	84.71
precosat	1693	772	587	4	0	73.67	87.72

Table 46: fuzz with all mutators

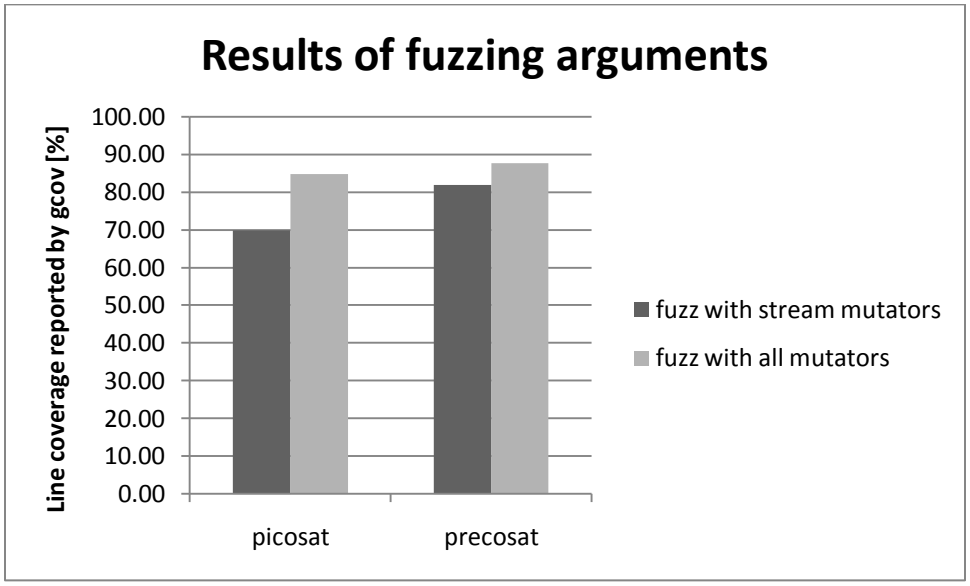

Figure 29: fuzz with all mutators

2.17.14 Conclusion of extending mutators

After adding mutators to further increase the coverage a fair quantity of additional code was covered by the fuzzing application. Table 47 gives some details of the amount of covered code comparing the cnfuzz tool, a version of the fuzzing application which generates solely valid formulas and the final version of the fuzzing application. Figure 30 compares the amount of covered code lines graphically.

time	60 min					
	cnfuzz		fuzz valid formulas only		fuzz	
solver	#tests	coverage	#tests	coverage	#tests	coverage
picosat	43138	67.97	1913	67.93	2816	84.71
precosat	36412	81.11	1806	82.40	1693	87.72

Table 47: Overview of tested fuzzers

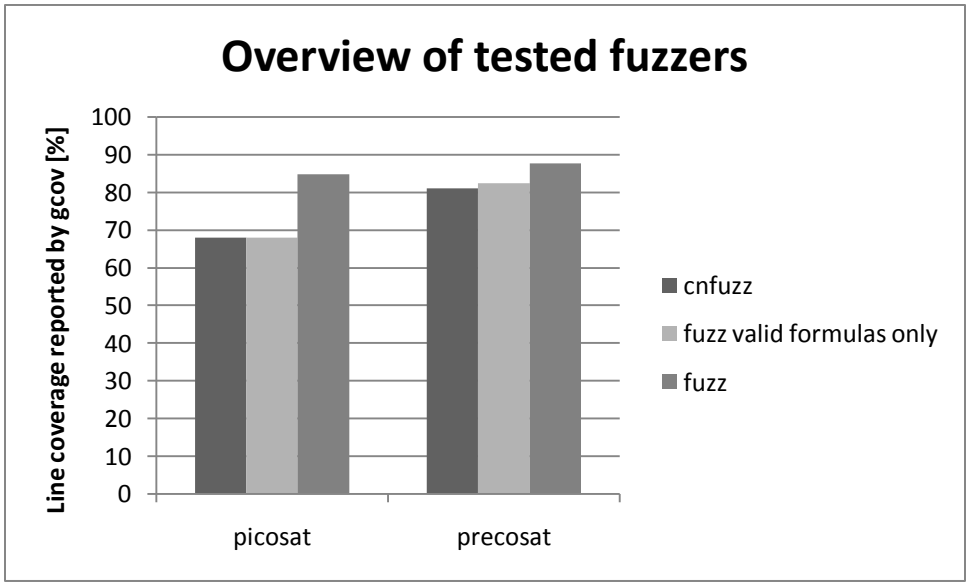

Figure 30: Overview of tested fuzzers

2.18 Final tests

This section gives details about a final test carried out to check if the amount of coverage will be further increased if the fuzzing application is used for a time of three hours. Details of the test run are given in Table 48. The amount of the gain in coverage is also depicted in Figure 31. As can be seen in the diagram the amount of coverage could be further increased by an amount of 1.06 percentage points in the case of picosat and 0.84 percentage points in the case of precosat. Another purpose of this test was to generate a representative test suite as described in the next section.

time	180 min						
	Tests						
solver	#tests	#sat	#unsat	#timeout	#undefined	byte coverage	linecoverage
picosat	8282	2891	658	125	919	77.91	85.77
precosat	6045	2540	1672	221	8	74.54	88.56

Table 48: Testing for 180 minutes

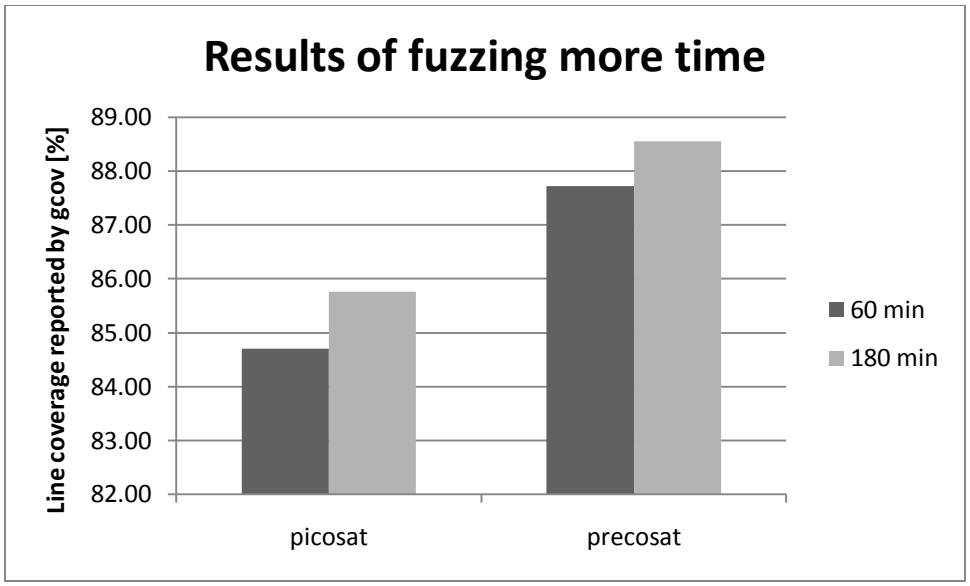

Figure 31: Testing for 180 minutes

2.19 Generation of the test suite

As described in Sec. 2.5 the final fuzzing application is capable of providing a test suite that covers all code locations that were executed during the fuzzing process in an output directory which can be specified by an argument to the fuzzing application. Basically this suite is generated by traversing all available basic blocks and serializing the most suitable input for each basic block to a file. To keep the test suite small duplicate inputs are omitted during the traversal. Additionally a script file is generated which contains all commands to execute these inputs with the necessary arguments. The script file is custom to the used operating system and details about the layout of a script file are given in Sec. 6.9. Besides these inputs, inputs that yielded anomalous behavior are stored in the output directory too. These are inputs that caused the target application to return an unspecified value, inputs that caused an exception as described in Sec. 2.5.4 and inputs that timed out. Finally the latest version of the coverage bitmap, as described in Sec. 2.5.6 is also saved in the output directory.

2.19.1 Usages of the test suite

The generated inputs in the test suite can be used in many ways to perform additional tests. First of all the inputs which caused an assertion, in case some are available, should be considered in detail. In the case of testing the picosat solver as described in Sec. 2.18 18 inputs that raised a FATALSIG were generated whereas all of them uncovered the same issue in the solver which was caused by calling abort. In the case of precosat no inputs were generated that raised a signal. All inputs that returned an unexpected value should be considered in more detail too. If the targeted solving application is designed to return a certain value in case of an exceptional behavior these values could be used to identify issues. Additionally certain methods of the runtime library, such as abort or assert, are designed to return special values to the caller

in case they are executed which should be considered in more detail too. However applications are often designed to not call such functions in case they are compiled highly optimized. Nevertheless a simple attempt is to replace the solving application by a version which has check code and debugging information enabled. Now all inputs which target the code in the optimized version of the solver can be processed again by the debug version of the solver by calling the generated script files. Using this attempt is was possible to identify four issues when the precosat solver was tested. Additionally tools such as gdb or valgrind could be used to observe the solving application. (Zeller, 2006) gives some detailed information on available tools and their usage regarding this task. Tools such as gcov could be used to identify code which was not executed by the test suite. This information may be of interest in order to identify unreachable code. If the targeted fuzzing application is available on different platforms the generated test suite could be cross checked on them. Test suites generated on a Linux based system using the highly optimized versions of the targeted solving applications were used to execute debug versions of the SAT solvers on a Windows based system. With this attempt one issue could be identified which concerns both solvers. Finally another interesting test, especially targeting the performance of SAT solvers, is to use all inputs that timed out, which are available in the test suite, and compare the times of different SAT solvers processing these inputs. With this test it was possible to identify an input that can be solved fast by the precosat solver but caused a timeout if it was processed by the picosat solver. Sec. 3.1 gives further details about the individual bugs found in each of the tested solving applications.

3 Final thoughts

This and the following sections contain some final thoughts about the developed and tested fuzzing application. The first finding of this thesis is that it is technically possible to fuzz a highly specialized application in an automated way whereas feedback is used to generate tests of high quality. Nevertheless the amount of covered code lines when only valid formulas were generated could not be increased as much as thought at the beginning of this thesis, compared to using the cnfuzz tool. This may draw the conclusion that quality cannot compensate quantity in the case of fuzzing SAT solving applications. However the final amount of covered code after adding mutators that operate on the input stream and the arguments is very promising. Besides the amount of automatically covered code a number of issues in the tested SAT solvers were detected which are described in more detail in the following sections. The bugs were reported and are fixed in the current versions of the solvers. Therefore this thesis succeeded in improving the quality of the tested SAT solving applications which is a great success.

3.1 Found Bugs

The following sections describe some of the issues that were found during the development and testing of the fuzzing application.

3.1.1 Bugs found by porting the solvers

Besides bugs found by the fuzzing application some issues were detected during the porting of the targeted solving applications. As already mentioned the fuzzing application was developed using Microsoft's .Net Framework. In order to test the fuzzer also on Windows based operating systems porting of the solvers was necessary. During this attempt two issues were identified which are related to using different compilers and runtime environments. One issue was caused by using the sizeof operator with an array containing no elements. The behavior of this operation is not clearly defined in the C standard and the used compilers reported different results of this operation. Another issue has been detected when values were passed to certain functions of the C runtime library as the runtime library primarily used to make the solvers did not exactly check the validity of the arguments. The issue was uncovered in the ported version of the solver if unsigned character values higher than or equal to 128 were passed to functions expecting positive signed character values.

3.1.2 Bugs found by analyzing the solvers

During the tests of mutators which operate on arguments minor incompatibilities between arguments reported by the usage information of the picosat solver and the arguments actually used were detected.

3.1.3 Bugs found in the picosat solver with the fuzzing application

Further issues which are specific to the picosat solver were detected by the fuzzing application and reported as assertions in the test suite. In case the picosat solver is instructed to generate a

proof of the satisfiability of a given problem and verbose output is enabled the solver aborts in case the given problem is unsatisfiable. As this raises a FATALSIG it was detected by the fuzzing application and a whole number of inputs that causes this signal were created in the test suite. Additionally the fuzzing application generated a certain test in the test suite that could not be solved within the given amount of time by the picosat solver but could be solved quickly by the precosat solver.

3.1.4 Bugs found in the precosat solver with the fuzzing application

During the tests of the precosat solver a number of issues which were detected by the execution runtime as assertions were found. One of the issues was expected to be found as an early version of precosat that contained a well known bug was tested in order to check if the fuzzing application is capable of detecting it. Besides this expected issue three additional issues were found in a current version of the precosat solver which could be fixed. Nevertheless one bug was still found in the fixed version of the precosat solver that will be fixed in future versions.

3.2 Given enough time

Given enough time the fuzzing strategy implemented in the fuzzing application should be at least as successful regarding code coverage as the strategy of generating random inputs solely. This is based on the fact that generating completely random inputs is part of the available mutators and will be used randomly. Nevertheless some parts of the picosat solver were not executed during the tests given an equal amount of time as described in Sec. 2.16.1. Therefore there is definitely space for further improvements like creating mutators that are more specialized on the problem of formulas in CNF. The drawback of using more specialized mutators is that the overall flexibility is reduced and more optimizations concerning the target application are necessary. Another drawback is that mutators which give good results quickly often fail to further improve the results if more time is given which is known as premature convergence (Wikipedia, 2009) in the field of genetic algorithms. The following sections give some information on other ways to further improve the developed fuzzing application.

3.2.1 Improve the search strategy

Although many tests were carried out in order to find optimal parameters for the fuzzing application only a small amount of all possible configurations were examined throughout this thesis. As there are strong correlations between the many parameters to the various steps in the fuzzing progress a lot of further testing would be required to really optimize the whole parameter set. Additionally there are correlations between parameters and the targeted application. Therefore it would be necessary to optimize the parameter set for each tested target application independently. Further improvements are possible during the monitoring step. It would be possible to not only check which basic blocks were executed but to

additionally get information about the way the final control instruction at the end of a basic block determines the next basic block. For instance, if the control instruction decides based on the outcome of a compare operation the amount of the difference of the compared variables could be used as information to check whether the search for inputs is directed into the desired direction. The big problem with this approach is that there is no direct correlation between compare and jump instructions in the case of x86 binary code. Therefore it would be necessary to manage information about every instruction that can change the state of the flags of the CPU which would make the monitoring step very expensive.

3.2.2 Speed up the fuzzing process

This section discusses some thoughts about speeding up the fuzzing process to carry out more tests within the given amount of time. By far the most of the time is consumed in the monitoring step which is caused by a limitation of this thesis to not depend on source code of applications. If the source code of applications is available it would be possible to instrument the code during compilation and use tools such as gcov to get the trace information of application runs. The big advantage of gcov is the very low impact on the runtime of the target application. It has been used throughout this thesis to check the validity of generated data and it exposed that the running time of applications instrumented to be used by gcov is about 20 times faster than the same application instrumented with pin. Another possibility to speed up the monitoring step would be to distribute the load to multiple machines which is described in the next section.

3.2.3 Distribute the load of monitoring to multiple machines

Another possibility to speed up the fuzzing process in its current form would be to distribute the load of monitoring to several machines as depicted in Figure 32. As shown in the figure one way to do this would be to have the fuzzing logic on one machine and the generation of the trace information through the monitoring tool on multiple machines. The communication could be implemented based on the producer consumer paradigm in a way that the fuzzing logic generates inputs based on the current application state and every input is sent to one of many clients. The clients run the target application with the supplied input using pin to generate the trace which is sent back to the fuzzing logic where it is further processed.

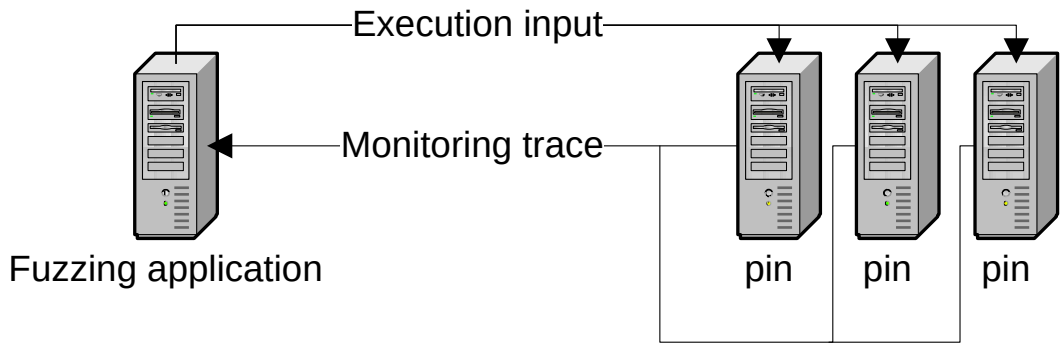

Figure 32: Distribute monitoring

3.3 Porting the fuzzer to other applications

Finally this section shows steps that would be necessary to use the developed fuzzing application to test arbitrary applications. As the fuzzing application pursues a pure black box fuzzing strategy any application, that takes some kind of mutable input and or uses arguments, could be fuzzed in principal. The whole monitoring step and the selection of inputs and mutators can be used without modification. The only thing that requires application dependant implementations are the actual mutators including the mutator to generate initial inputs. If it is necessary to port the fuzzer to target other applications a clearly defined interface between the classes responsible for mutation and the rest of the application could be defined. This would make it possible to implement the mutator interface e.g. by loading a third party library and all there is left to do for a third party user of the fuzzing application to test custom applications would be to provide an implementation of the mutator interface.

4 Conclusion

The main topic of this thesis was to check if it is possible to find issues in SAT solvers if the amount of code covered during the fuzzing progress is maximized. The thesis showed that it was indeed possible to find problems in the solvers although they were used and tested for quite some time. Therefore the attempt to increase code coverage during fuzzing proved to be a valuable strategy in order to find bugs in software.

Additionally the fuzzing application clearly succeeded in generating inputs of high quality which is given by the fact that only about $1/20^{th}$ of the number of test inputs were generated to cover equally many or even more parts of the tested SAT solving applications compared to using purely random inputs. This is concluded by comparing the amount of covered code lines of the tests generated by the fuzzing application with the amount of code lines covered by tests created with the cnfuzz tool which proved to be very well suited for generating random formulas. This may be a reason why the absolute amount of additionally covered code lines was

rather small. The detailed analysis of code that was not covered by the fuzzing application, as described in Sec. 2.16.1, showed that sometimes very large inputs would be necessary to enter several paths in the tested applications. In the tests carried out to compare the results with the cnfuzz tool such large inputs were prohibited as the time to process a single input was limited in order to increase the overall number of tests carried out by the fuzzer.

Nevertheless the amount of covered code could not be increased as much as expected at the beginning of this thesis. As already mentioned the main issue of the technique examined throughout this work is the additional time that it takes to get coverage information which nearly compensates the advantage of generating inputs of high quality. This is especially caused by monitoring the execution of the target application. Sec. 3.2 shows some possibilities to overcome this problem, e.g. if methods were used to instrument code during compilation the penalty of getting coverage information could be decreased a lot. But as this work pursued a black box fuzzing strategy a tool such as pin was necessary to get coverage information and although getting the required information with pin is many times faster than with other tested technologies it is mainly responsible for the additional time required to process inputs. Possibilities to get the desired information faster can be expected as faster virtualization and execution methods may be available in the future. Another problem is that only a small set of the huge number of possible configurations, especially of the mutators, were tested. They were mainly optimized to generate suitable inputs within a small number of uses. As observed many times in the field of evolutionary algorithms mutators which quickly give good results often fail to further improve results if more time is given. But as the time to fuzz the SAT solvers, and therefore the number of test inputs that were generated by the fuzzer, was limited it was necessary to configure the mutators to give valuable results quickly.

Generally, the constraint to limit the testing time was chosen mainly to compare several fuzzing strategies. As shown in Sec. 2.18 the fuzzing application developed throughout this thesis is capable to further increase the amount of covered code parts if more time is given to the fuzzing progress which is also true if the timeout specified per input would be increased. On the other hand if the attempt of creating completely random input is pursued for a longer time, chances that further parts of the tested application are covered are rather low.

Another important finding of this thesis is that it is important to not only test valid inputs but to also fuzz arguments which are passed to the SAT solving applications and to implement methods to generate invalid inputs. These two facts were not considered in previous attempts to fuzz the tested solvers and by implementing them not only the amount of code coverage could be increased significantly but also some issues were found in the fuzzed applications. This helped to improve the quality of the tested software and can be seen as an unexpected, but great success.

If the developed fuzzer is modified to universally test applications, which is basically possible for all kinds of software that take some kind of mutable input and or use arguments, a handy tool to automatically generate test inputs is provided. As described in Sec. 2.19.1 a great number of options are available to use these inputs in order to improve the quality of software. Therefore besides proving that it is technically possible to implement a fuzzing application which tries to maximize code coverage using black box fuzzing a set of valuable methods to improve fuzzing in general were introduced throughout this thesis. And if methods are available to get coverage information faster the process of testing the quality of software can be improved significantly.

5 Bibliography

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice.* New York: Oxford University Press.

Bäck, T., & Schwefel, H.-P. (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation , 1*.

Biere, A. (2009). P{re,i}coSAT@SC'09.

Biere, A. (2009). *Software*. Retrieved from http://fmv.jku.at/software/index.html

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. K. (1990). Occam's RAZOR. *Readings in Machine Learning* .

Box, G. E., & Muller, E. (1958). A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics , 29* (2).

Devroye, L. (1986). *Non-Uniform Random Variate Generation.* New York: Springer-Verlag.

DIMACS Challange. (1993, May 8). *Satisfiability Suggested Format*. Retrieved from ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/

GNU Free Software Fundation. (2008). *gcov - a Test Coverage Program*. Retrieved from http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

Granneman, S. (2006, July). *A Month of Brower Bugs*. Retrieved from http://www.securityfocus.com/columnists/411

Intel Coporation. (2009). *Pin*. Retrieved from http://www.pintool.org/downloads.html

Intel Corporation. (2009). *Pin 2.7 User Guide*. Retrieved from http://www.pintool.org/docs/29972/Pin/html/

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., et al. (2005). Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)* , pp. 190-200.

Microsoft Corporation. (2009). *msdn Microsoft Developer Network*. Retrieved from http://msdn.microsoft.com/en-us/library/default.aspx

Miller, B. P., Fredrikson, L., & Bryan, S. (1990). An empirical study of the reliability of UNIX utilities. *Communications of the ACM Volume 33* , pp. 32-44.

Miller, B. P., Koski, D., Pheow, C., Maganty, L. V., Murthy, R., Natarajan, A., et al. (1995). *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services.* University of Wisconsin-Madison.

Mitchell, D., Selman, B., & Levesque, H. (1992). Hard and Easy Distributions of SAT Problems. *Proceedings AAAI-92* , pp. 459-465.

Sparks, S., Embleton, S., Cunningham, R., & Zou, C. (2007). Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. *Computer Security Applications Conference, Annual* .

Takanen, A., Demott, J. D., & Miller, C. (2008). *Fuzzing for Software Security Testing and Quality Assurance.* Norwook, MA 02062: ARTECH HOUSE, INC.

van Sprudel, I. (2005). *Fuzzing: Breaking software in an automated fashion.* 22nd Chaos Communication Congress.

Wikipedia. (2009). *Basic Block*. Retrieved from http://en.wikipedia.org/wiki/Basic_block

Wikipedia. (2009). *Common Intermediate Language*. Retrieved from http://en.wikipedia.org/wiki/Common_Intermediate_Language

Wikipedia. (2009). *Executable and Linkable Format*. Retrieved from http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Wikipedia. (2009). *Infinite monkey theorem*. Retrieved November 26, 2009, from http://en.wikipedia.org/wiki/Infinite_monkey_theorem

Wikipedia. (2009). *Premature convergence*. Retrieved from http://en.wikipedia.org/wiki/Premature_convergence

Wikipedia. (2009). *Tower of Hanoi*. Retrieved from http://en.wikipedia.org/wiki/Tower_of_Hanoi

Zeller, A. (2006). *Why programs fail.* Germany: dpunkt.verlag.

6 Appendix

6.1 Configuring the fuzz application

Basically there are two possibilities to influence the way the fuzzing application operates. The first one is given by passing arguments to the fuzzing application. Table 49 shows a list of available arguments and a short description of each. The second option is to modify values in the application configuration file. This file is named fuzz.exe.config and resides in the same directory as the fuzzing application. It contains a list of configuration settings in XML format. Complete information of the available settings as well as a short description of each is given in Table 50.

Argument	Default value	Description
<application>		The target application to be fuzzed
-h		Show the usage screen
-m		Total fuzzing time
-t	5 sec	Timeout for target application in seconds
-a		Arguments file
-o	out	Output directory
-s	0	Seed value for random generator

Table 49: Arguments to the fuzzing application

Setting	Default Value	Description
VisDisplayForm	True	Display the visualization form
VisOutputColoring	True	Use different colors in the output
VisPrintBBLStatistics	True	Display statistics about basic blocks in the output
VisPrintMutatorStatistics	True	Display statistics about mutators in the output
RetValSat	10	Return value of satisfiable instances
RetValUnsat	20	Return value of unsatisfiable instances
RetValAccepted	0	Additional return values which should be accapted by the fuzzing application
RandNewNumVariablesMean	50	Mean amount of variables in a random formula
RandNewNumVariablesDeviation	40	Deviation of variables in a random formula
RandNewNumClausesFactorMean	13.5	Mean ratio between the number of clauses and the number of variables in a random formula
RandNewNumClausesFactorDeviation	3	Deviation of the mean ratio between the number of clauses and the number of variables in a new formula
RandNewNumLiteralsMin	3	Minimal number of literals in clauses of a random formula
RandNewNumLiteralsExpMean	4	Exponential mean of the number of additional literals in a clause of a random formula
RandNewTargetVariableDeviation	18	Deviation of variables in a clause of a random formula
MutatorAddClausesMean	750	Exponential mean number of clauses that are added by the mutator AddClause
MutatorDeleteClausesMean	750	Exponential mean number of clauses that are deleted by the mutator DeleteClause
MutatorAddLiteralMean	500	Exponential mean number of literals that are added by the mutator AddLiteral
MutatorDeleteLiteralMean	500	Exponential mean number of literals that are deleted by the mutator DeleteLiteral
MutatorSwapSignMean	500	Exponential mean number of signs that are swapped by the mutator SwapSign

MutatorSwapVarMean	500	Exponential mean number of variables that are swapped by the mutator SwapVar
MutatorSwapVarDeviation	100	Deviation of the value of new variables that are modified by the mutator SwapVar
MutatorFitnessBonusUpdateBBL	1	Amount of fitness increment in case a mutator succeeded in updating a basic block
MutatorFitnessBonus	10	Amount of fitness increment in case a mutator succeeded in detecting or executing new basic blocks
BblTournamentSize	15	The size of the tournament used to select basic blocks
BblUseBasicBlockFitness	False	Specifies if the fitness of basic blocks should be considered in case the difference of available inputs is too small
BblFitnessBonus	3	Amount of fitness increment in case a basic block was used successfully
SelBblMaxSearchOffset	20	Amount of nearby addresses that are searched for a suitable basic block
SelBblMinNumInputsDiff	1	Minimal difference of the number of available inputs of a basic block to be considered fitter
ModifyCharMean	50	Exponential mean number of characters which are changed by the mutator ModifyChar
ModifyCharDeviation	3	Deviation of the value of new characters which are modified by the mutator ModifyChar
AddWellKnownPhraseMean	10	Exponential mean amount of new phrases that are added by the mutator AddWellKnownPhrase
DuplicateCharMean	20	Exponential mean number of characters that are duplicated by the mutator DuplicateChar
DeleteCharMean	20	Exponential mean number of characters that are deleted by the mutator DeleteChar
MaxSearchForInputWithArguments	100	Number of times a search is restarted in case an input with arguments is required but not found
DeleteArgumentMean	1	Exponential mean number of arguments that are deleted by the mutator DeleteArgument
DeleteArgumentCharMean	1	Exponential mean number of argument characters that are deleted by the mutator DeleteArgumentChar
DuplicateArgumentCharMean	1	Exponential mean number of argument characters that are duplicated by the mutator DuplicateArgumentChar
ModifyArgumentCharMean	3	Exponential mean number of argument characters that are modified by the mutator ModifyArgumentChar
ModifyArgumentCharDeviation	5	Deviation of the value of new argument characters which are modified by the mutator ModifyArgumentChar

Table 50: Settings in the configuration file

6.1.1 Notes on the seed value

Although a seed value can be specified in the command arguments to the fuzzing application the whole fuzzing progress is not deterministic. This is caused by several additional circumstances, such as the speed of the testing machine, that influences whether processing of an input times out. As it is not defined which inputs actually raise a timeout but the fuzzing process depends on them the whole process becomes nondeterministic. The main purpose of the seed value is to provide the possibility to start the fuzzing process with different initial values if it is run on several machines. That way the diversity of the overall tested inputs will be increased if the fuzzing application is run on multiple machines in parallel with different seed values.

6.2 Installation on Linux based systems

This guide gives some instructions on how to install the fuzzing application on Linux based systems. In the test installation Ubuntu Linux in the version 9.10 was used.

6.2.1 Installation of pin

The files required to install the pin tool are available at (Intel Coporation, 2009) and need to be extracted to a local folder, e.g. ~/pin. To run the pin tool it is necessary to export the path to the pin script to the PATH environment variable, e.g. 'export PATH=$PATH:~/pin'. To check if the pin tool is working it can be run from the command prompt by launching 'pin'.

6.2.2 Making the sat solver

To make the satisfiability solvers used throughout this thesis it is necessary to download the source files, available at (Biere, Software, 2009), to a local directory. They can be build by running './configure' and 'make'. In the case of picosat it is advisable to remove the '-static' option. Otherwise the C runtime library is linked statically with the picosat solver and the fuzzing application would additionally try to fuzz the C runtime which would drop the fuzzing performance dramatically. If coverage information should be provided in the application the compiler flags '-fprofile-arcs' and '-ftest-coverage' need to be added in the corresponding makefiles. If they are present line coverage information could be determined by running the gcov utility (GNU Free Software Fundation, 2008).

6.2.3 Making the Monitor tool

To make the monitor library it is necessary to change to the directory of the MonitorTool and launch the build process, e.g. 'make PIN_HOME=~/pin'. Making the tool requires the g++ compiler which is available via, e.g. 'apt-get install g++'. Assuming picosat is used as sat solver the Monitor tool can by tested by running 'pin –t Monitor.so -- ./picosat –h'. If everything worked fine the file trace.xml is generated and contains information about the application run as described in Sec. 2.3.

6.2.4 Installation of the .net runtime

As the fuzzing application is written using the .Net framework a .Net runtime is required. In the case of Ubuntu the mono runtime can be installed, e.g. by running 'apt-get install mono-runtime'. The fuzzing application additionally requires win forms to display coverage information. By default this library is not installed but can be added e.g. by running 'apt-get install libmono-winforms2.0-cil'.

6.2.5 Installation of the fuzz tool

The fuzzing application is installed simply by copying the required files to a local directory. To check if the fuzzer is working it can be tested by running 'mono fuzz.exe'. If everything works the usage information of the fuzzing application will be prompted.

6.3 Detailed description and code excerpts of the monitor tool

This section gives some details about the implementation of the monitor library. As it is the case with ordinary applications written in the C language the executions starts with the main

method which is depicted in Figure 33. In contrast to ordinary applications the code in this function is not executed directly by the CPU but is already run in a virtual machine provided by the pin tool. After some general initialization the method sets up callback functions which are called when certain events occurred, such as loading of an image or unveiling a new trace. Finally the execution of the target program is initiated by calling PIN_StartProgram. All instrumentation functions are implemented in a very similar way and therefore only the implementation of the TraceCallback function is described as an example in more detail in the following section.

```
int main(int argc, char * argv[])
{
  // Initialize pin
  if (PIN_Init(argc, argv))
  {
    return Usage();
  }

  // instrument image load
  IMG_AddInstrumentFunction(ImageLoad, 0);

  // instrument trace
  TRACE_AddInstrumentFunction(TraceCallback, 0);

  // instrument context change (DebugEvents caused by exceptions, ...)
  PIN_AddContextChangeFunction(OnException, 0);

  // exit handler
  PIN_AddFiniFunction(Fini, 0);

  // start the program
  PIN_StartProgram();

  return 0;
}
```

Figure 33: main method of the monitor library

6.3.1 Implemention of the TraceCallback function

This function is provided by the monitor library and called every time a new trace is detected by the pin tool. Figure 34 shows the main parts of the implementation of this function. Basically all static basic blocks in the trace are traversed and some general information is stored in a list. Additionally a callback function is added to each basic block which will be executed every time the basic block is executed. The callback function takes the address of the execution counter of the static basic block in the list as argument. Some implementation details of the BBLCallback function are given in the next section.

```
VOID TraceCallback(TRACE trace, VOID *v)
{

  // foreach BBL
  for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
  {
    ADDRINT address=BBL_Address(bbl);
    if (address>=imageLowAddress && address<=imageHighAddress)
    {
        BBLCountEntry *pBBLCountEntry=new BBLCountEntry;
        pBBLCountEntry->addr=address;
        pBBLCountEntry->unSize=BBL_Size(bbl);
        pBBLCountEntry->ulCount=0;
        BBLCounter.push_back(pBBLCountEntry);

        BBL_InsertCall(bbl, IPOINT_ANYWHERE, AFUNPTR(BBLCallback), IARG_FAST_ANALYSIS_CALL,
                       IARG_PTR, &(pBBLCountEntry->ulCount), IARG_END);
    }
  }
}
```

Figure 34: Implemetation of the TraceCallback function by the monitor library

6.3.2 Implementation of the BBLCallback function

This function is called every time the specified basic block is executed. As already mentioned in the previous section it takes the address of the execution counter of the static basic block as argument. All it does is to dereference the pointer and increment the value of the pointer by one as shown in Figure 35.

```
VOID PIN_FAST_ANALYSIS_CALL BBLCallback(void* pArg)
{
  UINT64* pCounter=(UINT64*)pArg;
  ++(*pCounter);
}
```

Figure 35: Implemention of the BBLCallback function

6.4 Killing an application

If the tested application does not quit within a specified amount of time it will be terminated by the fuzzer. In a first attempt the application will be killed softly by raising the SIGTERM signal if the application is run on Linux based systems or by sending CRTL-C if the system is run on Windows based systems. If the application still doesn't quit within a certain amount of time the corresponding process is terminated by killing it via the System.Diagnostics.Process API.

6.5 Creating non-uniform distributed random numbers

In many programming languages only methods to generate uniquely distributed random values are available. To generate values that are non-uniformly distributed some additional methods needed to be implemented. In the case of Gaussian distributed random numbers used throughout this thesis an algorithm known as Box-Muller transform (Box & Muller, 1958) was used. Exponentially distributed random numbers were generated based on inverse transform sampling (Devroye, 1986).

6.6 Monitoring performance analysis

The following sections highlight some issues that were encountered during the development of the monitor library.

6.6.1 Problem that the processor's single step is too slow

To test the performance of monitoring a simple test application that solves the Tower of Hanoi problem (Wikipedia, 2009) was implemented. The reason for using such a test application was that it is a very short application which easily allows long processing times as the problem grows exponentially with the number of disks. The diagram depicted in Figure 36 shows that it is possible to solve problems with up to 20 disks within one second if the application was started natively. Figure 37 shows that solving the problem with 15 disks took more than 500 seconds if the Single Step mode of the CPU was used. The ratio between the time it takes when using the Single Step mode and native execution is between 7000 and 20000 which makes using the Single Step mode far too slow for using it to monitor the fuzzing progress.

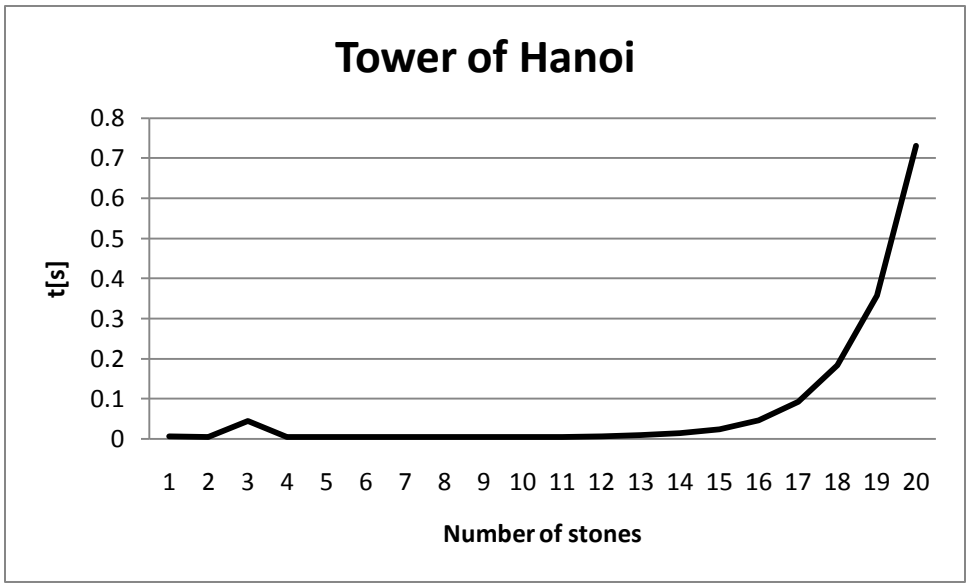

Figure 36: Tower of Hanoi

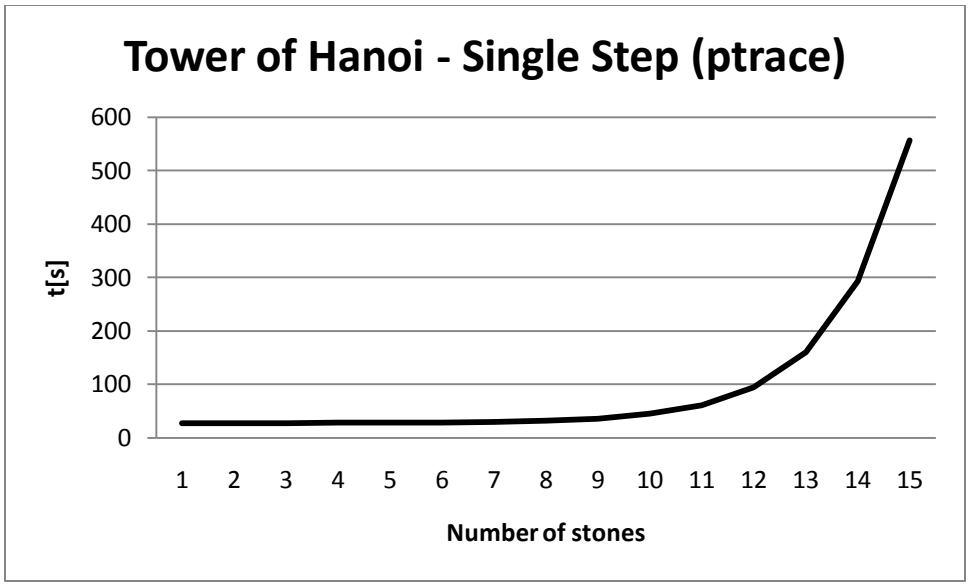

Figure 37: Tower of Hanoi - Single Step

6.6.2 Performance of generating the exact application trace with pin

This test uses pin and a version of the monitor library that generates an exact trace of the application run. As the results, depicted in Figure 38, show it is possible to solve problems with up to 20 disks. The ratio between the times it takes when using the pin tool and normal execution is about 36 at the beginning and decreases as the problem gets harder as depicted in Figure 39. This is explained by the fact that the number of statically available basic blocks of the tested application is very small and pin caches and reuses information gained about basic blocks.

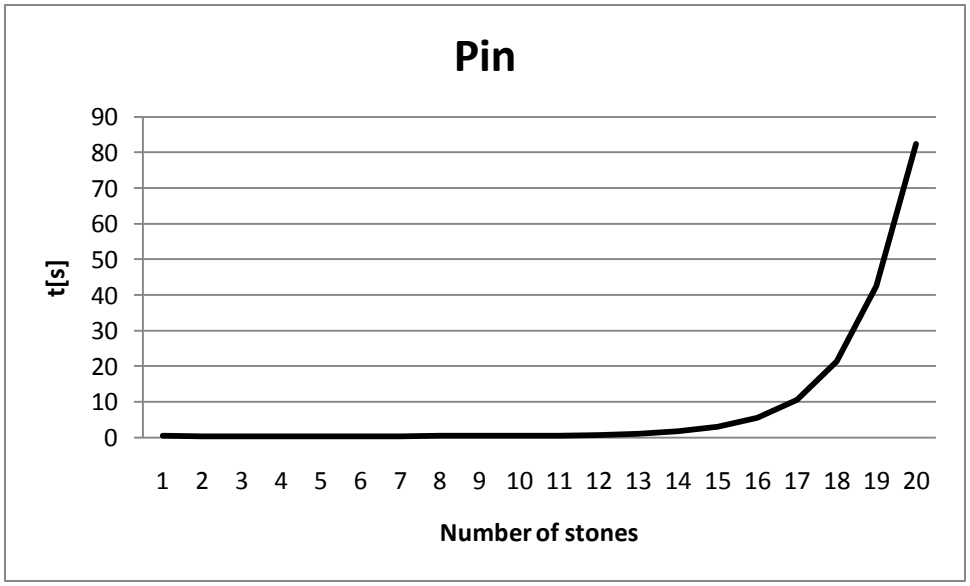

Figure 38: Tower of Hanoi – pin

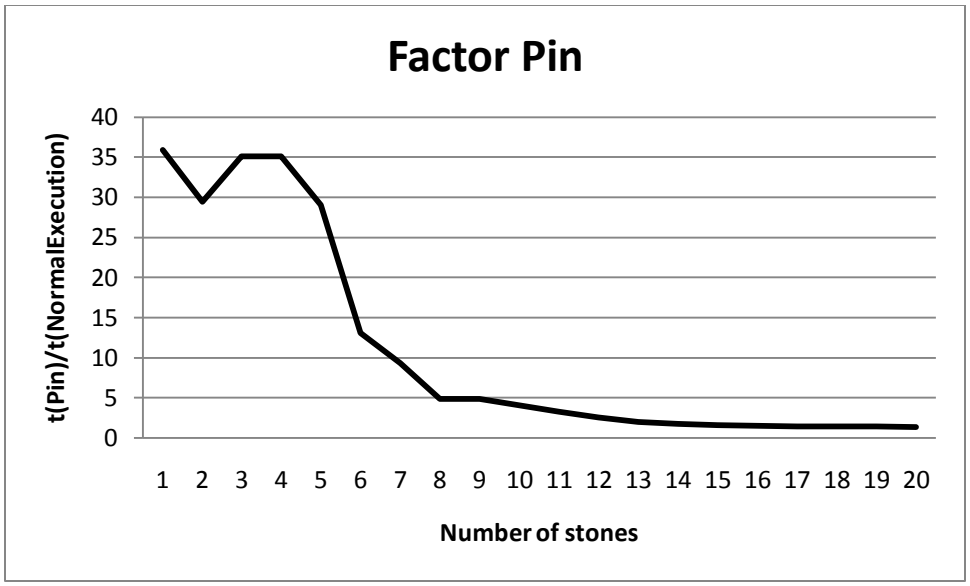

Figure 39: Tower of Hanoi Performance ratio - pin

6.6.3 Counting BBLs on picosat and precosat

Using cnfuzz with seed 362 generates a rather hard formula which is made of 4053 clauses and 552 variables. Running pin with inscount2.so, an example delivered with pin, shows that picosat needs 785662240 instructions (124053032 basic blocks) to solve it. Running the same input with precosat unveils that it takes even 1513912100 instructions (238697973 basic blocks) to solve the formula. Storing a minimal amount of 4 bytes per executed basic block would require about 911 MB of memory.

6.6.4 Using the monitor tool with picosat

Figure 40 gives details of a test that was carried out which used the pin tool with the monitor library and executed the precosat solver with a number of different inputs. The test showed that the average amount of time necessary to get trace information from the precosat solver is about 1.2 seconds.

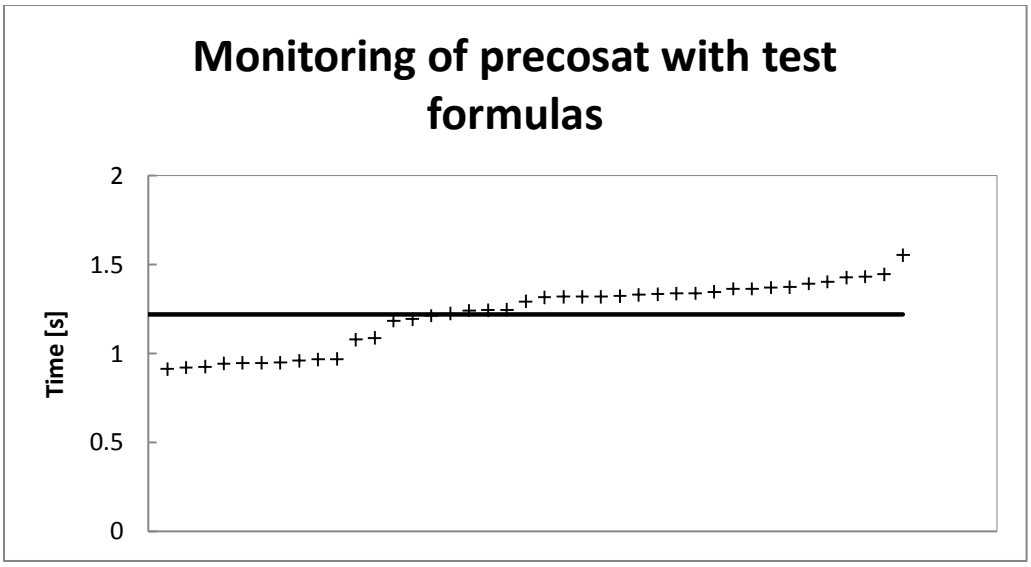

Figure 40: Monitoring precosat with test formulas

6.7 Bash script to run cnfuzz

Figure 41 shows the bash script that was used to run cnfuzz for a certain amount of time and get coverage information after every run. The script takes two parameters. The first one specifies the amount of time the script is run in minutes and the second one specifies the SAT solver application to use. The script counts the number of tests executed and calls gcov after every test for each available source file. The collected gcov information is then appended to corresponding files which were used to generate the graphics in Sec. 1.6.

```
#!/bin/bash
START=$(date +%s)
END=$START

CNT=0
while [ $[ $END - $START ] -lt $[ $1 * 60 ] ]
do
 CNT=$(expr $CNT + 1)

 ./cnfuzz | $2

 echo "test $CNT"

 for srcfile in *.c
 do
  if [ -f $srcfile ]
  then
    echo "test $CNT" >> $srcfile.gcov.txt
    test -f $srcfile && gcov $srcfile >> $srcfile.gcov.txt
  fi
 done

 for srcfile in *.cc
 do
  if [ -f $srcfile ]
  then
    echo "test $CNT" >> $srcfile.gcov.txt
    test -f $srcfile && gcov $srcfile >> $srcfile.gcov.txt
```

```
  fi
 done

 END=$(date +%s)
done

END=$(date +%s)
DIFF=$(( $END - $START ))

echo "$CNT tests in $DIFF seconds"
echo "$CNT tests in $DIFF seconds" >> gcov.txt
```

Figure 41: Bash script to test cnfuzz

6.8 Description of the test environment

The development of the fuzzing application was done on a system which is shown in Table 51. All tests that were carried out throughout this thesis were made on a Linux based operating system as this is the system which is primarily targeted by the tested SAT solving applications. This system ran inside a virtual machine provided by Virtual PC 2007 on the development machine. Table 52 gives some details about the test system. To test the installation routines in Sec. 6.2 an alternative hardware system was used that had Ubuntu Linux 9.10 installed natively on the machine.

Processor	Intel® Core™2 Duo CPU E6750 @ 2.66 GHz
Memory	4 GB
Operating System	Windows 7 Ultimate
.Net Framework	Version 3.5 SP1

Table 51: Development system

Processor	Virtualized by Virtual PC 2007
Memory	512 MB
Operating System	OpenSUSE 11.1
.Net Framework	Mono 2.4.2.3

Table 52: Test system

6.9 Details about the generated test suite script file

The fuzzing application generates a script file which executes all inputs that are generated as documented in Sec. 2.19. The file depends on the used operating system and is generated as batch file in the case of a Windows based operating system and as shell script in case of a Linux based operating system. Figure 42 and Figure 43 show some excerpts of each of the files generated. Basically two lines are added per test which dump information of the current test and run the solving application with the specified input and arguments. The generation of these files is necessary to support testing of inputs with arguments.

```
#!/bin/sh

echo "./precosat < Coverage001.cnf"
./precosat < Coverage001.cnf

echo "./precosat < Coverage002.cnf"
./precosat < Coverage002.cnf
```

Figure 42: Generated script file – Linux

```
@echo off

echo "picosat.exe -n < Coverage005.cnf"
picosat.exe -n < Coverage005.cnf

echo "picosat.exe < Coverage006.cnf"
picosat.exe < Coverage006.cnf
```

Figure 43: Generated batch file - Windows

6.10 Analysis of cnfuzz

The algorithm basically creates formulas based on a model consisting of layers, clauses and literals as depicted in Figure 44. The number of layers is chosen from a discrete uniform distribution in the range of 1 to 20.

$$n \sim U(1; 20)$$

Each layer has an equally distributed width w_i between 10 and W which corresponds to the number of variables that belong to this layer. W is picked once for all layers in the range of 10 to 70.

$$W \sim U(10; 70)$$
$$w_i \sim U(10;\ W)$$

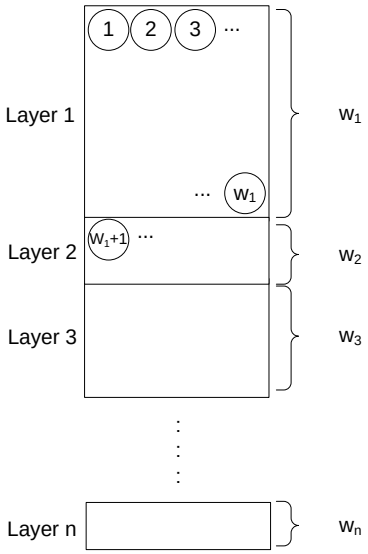

Figure 44: Layers of cnfuzz

After this setup each layer is processed one after each other and clauses are generated. The number of clauses c which is created for each layer is specified by

$$c \sim U(3; 4.5) * (w_i + w_{i-1})$$

The length l of each clause in a layer is distributed exponentially as

$$l \sim 3 + \exp(\frac{1}{3})$$

Finally the variables for the clauses are chosen from the current layer and its p previous layers.

$$p \sim \exp(\frac{1}{2})$$

$$v \sim U(w_{i-p}; w_i)$$

As depicted in Figure 45 the specified number of variables is chosen randomly from the specified layers (p=3 is used in the example picture), whereby duplicates in a clause are prevented. The sign of a variable is chosen equally distributed between true and false.

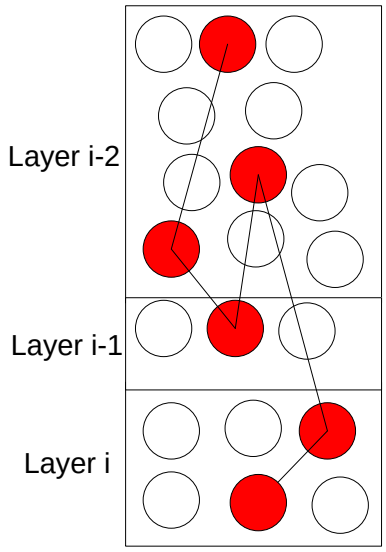

Figure 45: Selection of variables for layers in cnfuzz

7 Lebenslauf

Persönliche Daten

Name	Jürgen Holzleitner
Anschrift	Atzbach 132 / 7 4904 Atzbach
Geburtsdatum und –ort	12. Mai 1981 in Gmunden
Staatsangehörigkeit	Österreich
Familienstand	In Lebensgemeinschaft, 1 Kind

Schulische Ausbildung

1987 – 1991	Öffentliche gem. Volksschule Atzbach 4904 Atzbach
1991 – 1995	Hauptschule Schwanenstadt II 4690 Schwanenstadt
1995 – 2000	Höhere Technische Bundeslehranstalt Leonding 4060 Leonding
2003 – 2007	Johannes Kepler Universität Linz (Bachelorstudium Informatik) 4040 Linz

Berufliche Tätigkeiten

August 1998	STIWA Fertigungstechnik Sticht GmbH (Ferialpraktikum) 4800 Attnang-Puchheim
1999 – 2000	SWA Software Wizards GmbH (Auftragsarbeiten Softwareentwicklung) 4050 Traun
Seit 2000	SWA Software Wizards GmbH (Softwareentwicklung) 4050 Traun

8 Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplom- bzw. Magisterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benützt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Atzbach, am 24. Dezember 2009

Jürgen Holzleitner