TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Sicherheit in der Informatik

Dynamic Symbolic Execution with Scalable Interpolation Based Path Merging

Andreas Ibing

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender :	Univ.-Prof. Dr. Helmut Seidl
Prüfer der Dissertation :	
	1. Univ.-Prof. Dr. Claudia Eckert
	2. Univ.-Prof. Dr. Alexander Pretschner

Die Dissertation wurde am 23.11.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 04.04.2017 angenommen.

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der automatischen Erkennung von Fehlern (Common Weaknesses) in C Programmen. Der Ansatz ist eine selektive symbolische Ausführung auf der Quelltext-Ebene mit interpolationsbasierter Pfadverschmelzung. Interpolation und Pfadverschmelzung werden einerseits zur Verringerung der nötigen Rechenleistung ohne Genauigkeitsverlust der Fehlererkennung im Vergleich zu Path Coverage eingesetzt, erzielt wird dann ”Error and Branch Coverage”. Andererseits kann die Interpolation zur Erreichung eines wählbaren Coverage-Kriteriums (z.B. Branch Coverage) angepasst werden, um die nötige Rechenleistung noch weiter zur reduzieren, allerdings bei verminderter Genauigkeit der Erkennung. Die Interpolation besteht in einer Entfernung von Constraints aus dem Pfad-Constraint. Es wird gezeigt, dass die betrachteten Algorithmen eine Interpolations- und Coverage-Hierarchie ergeben, bezüglich Teilmengen von Constraints bzw. Teilmengen von Programmpfaden und Implikation von Coverage-Kriterien. Anpassungen des Quelltextes sind zur Analyse nicht erforderlich. Symbolische Ausführung mit Satisfiability Modulo Theories Solver wird mit Instrumentierung von Binärcode kombiniert, um verschiedene Fehlerarten effizient zu erkennen. Die Implementierung ist eine plug-in Erweiterung der Eclipse C/C++ Development Tools und verwendet ein Debugger Machine Interface. Sie wird mit Testprogrammen der Juliet Suite für Speicherzugriffsfehler, Endlosschleifen, Information Exposures, Dead Code und Race Conditions bewertet. Die vorliegende Arbeit basiert auf den auf Seite 109 folgende aufgelisteten Publikationen.

Abstract

This thesis deals with the automated detection of common software weaknesses in C programs. The approach is selective symbolic execution on the source code level with interpolation based path merging. Interpolation and path merging are used to reduce the necessary computational effort without reduction in bug detection accuracy compared to path coverage, the symbolic execution then achieves error and branch coverage. Interpolation is also used to further reduce the computational effort and to achieve a selectable coverage criterion (e.g., branch coverage), at the expense of false negative detections. The interpolation consists in a removal of constraints from the path constraint. It is shown that the considered algorithms yield an interpolation and coverage hierarchy with respect to subsets of constraints and subsets of program paths, and with respect to the implication of coverage criteria, respectively. Modification of source code is not necessary for analysis. Symbolic execution with Satisfiability Modulo Theories Solver is combined with binary instrumentation to efficiently detect different bug types. The implementation is a plug-in extension to the Eclipse C/C++ development tools and uses a debugger machine interface. It is evaluated by detecting memory access errors, infinite loops, information exposures, dead code and data race conditions in test programs from the Juliet test suite for program analyzers. The document at hand is based on the publications listed on page 109 and the following.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Common Software Weaknesses

The common weakness enumeration [*] (CWE) is part of a community effort for a common taxonomy of software bugs [MBC07]. It is closely related to the common vulnerabilities and exposures [†] (CVE) list, the common weakness scoring system [‡] (CWSS) and several other classifications. The common vulnerabilities and exposures are a set of unique identifiers for publicly known and potentially exploitable bugs. A bug is potentially exploitable, if it can be reached by data structures under control of an attacker. The US National Vulnerability Database [§] is a repository of the US government, that currently lists over 78000 CVE entries. The common weakness enumeration classifies different types of software bugs and can therefore be seen as an abstraction of the CVE. CVE entries are mapped to CWE types. The weakness enumeration is not a list, but rather a graph. It comprises different views, each with different abstraction levels. Category nodes comprise related sets of weaknesses. CWE node types are ’weakness base’, ’weakness variant’, ’weakness class’, ’category’, and ’view’. The CWE currently comprises 33 views, 244 categories, 719 weaknesses, and in total 1004 entries. An example view is ’development concepts’ (CWE-699), which comprises, e.g., the views ’weaknesses introduced during design’ (CWE-701) and ’weaknesses introduced during implementation’ (CWE-702). Another example is the weakness class ’improper restriction of operations within the bounds of a memory buffer’ (CWE-119), which, e.g., comprises the weakness base ’out-of-bounds write’ (CWE-787) and the weakness base ’access of uninitialized pointer’ (CWE-824). Most current program analysis tools adhere in their reports to the common weakness enumeration. The common weakness scoring system provides a risk metric to

[*] https://cwe.mitre.org

[†] https://cve.mitre.org

[‡] https://cwe.mitre.org/cwss

[§] https://nvd.nist.gov

prioritize software weaknesses. The prioritization can be used to decide which bug to fix first. The CWSS consists of the three factor groups ’base finding’, ’attack surface’ and ’environmental’. In sum, it comprises 16 factors. The base finding group comprises, e.g., ’technical impact’ and ’acquired privilege’. The attack surface group comprises, e.g., ’required privilege’. The environmental group comprises, e.g., ’likelihood of discovery’ and ’likelihood of exploit’. The 16 factors are scored, and the factor scores are combined in a formula to yield the CWSS score. The metric can be automatically computed by an analysis tool. In the following, the terms ’bug’ and ’error’ are used for brevity with the meaning of common software weakness in the sense of the CWE.

1.2 Software Testing and Coverage

The standard method to find and remove software bugs before deployment is to use a manually written test suite, i.e., to specify input and output vectors. For each input vector of concrete values, the program takes a certain path and produces a vector of concrete output values. The test suite detects a bug if the bug leads to unexpected output (including operating system error output). For systematic testing with a defined criterion for when to stop, a *code coverage* criterion can be used. During test suite execution, the execution coverage is then monitored. There are different coverage criteria, that consider the coverage of input values, control flow, data flow or thread interleavings.

An input coverage criterion is, e.g., parameter value coverage. It requires, that ’common’ values are covered for all input parameters of a method. ’Common’ values include boundary cases like empty or null input, or invalid input. Control flow oriented coverage criteria are, e.g., branch coverage or modified condition / decision coverage (MC/DC). Complete branch coverage means that every branch in the program is covered by at least one program path when executing the test suite. MC/DC is more comprehensive than branch coverage. It requires, that each control flow decision in the program takes every possible outcome and that each condition in a decision affects the decision outcome independently of the others. A data flow oriented coverage criterion is, e.g., define / use pair coverage. It monitors the data flow, which is given by variable definitions (writes) and variable uses (reads). For multi-threaded code, thread scheduling is additionally of interest. The detection of concurrency bugs like data races or deadlocks depends on the coverage of thread interleavings. A thread interleaving is a sequential execution of threads (a scheduling of all threads on a single CPU core). Data flow oriented coverage criteria can be extended to multi-threaded code. An example is local-or-remote-define coverage [LJZ07]. Complete coverage is achieved under this criterion, if the executions cover for each variable use both a thread interleaving in which the variable is defined thread-locally, and an interleaving in which the variable is defined by a different thread. Coverage criteria differ in complexity, i.e., in the number of program executions necessary to achieve the coverage criterion — and correspondingly in the size of the test suite. Of practical interest are branch coverage and MC/DC. Test suites for standard software typically target branch coverage, while for certain safety critical software, MC/DC is required.

Bug detection during testing can be improved by instrumenting the program with additional checks. The aim is to detect bugs on an executed path also in cases when the bug does not lead to unexpected output. Instrumentation can be performed on the source or on the binary level. During execution, the instrumentation traces relevant information and stores it in shadow memory and performs checks for bug detection. Source instrumentation automatically modifies source code. An example is `CCured` [NCH+05], that adds checks for type and memory safety to C programs. Additional checks can also be added by binary instrumentation. Binary instrumentation can be dynamic or static. Prominent frameworks for dynamic binary instrumentation are `Pin` [LCM+05] and `Valgrind` [NS07]. Example tools built with `Valgrind` are `Memcheck` [SN05] and `Helgrind` [MW06]. `Memcheck` detects memory access errors. `Helgrind` is a checker for data race conditions. It can detect a data race not only when it actually does occur with the current thread interleaving, but also when it might occur with a similar thread interleaving. Static binary instrumentation is often integrated into compilers (compiler instrumentation). Current compilers can, for example, add stack canaries to detect buffer overflows at runtime. Prominent checkers based on static binary instrumentation are `AddressSanitizer` [SBPV12], that detects memory access errors, and `ThreadSanitizer` [SI09], that detects race conditions. Both are integrated, e.g., in the GNU C Compiler (`gcc`). Instrumentation slows down execution. Compiler instrumentation typically has a lower overhead compared to dynamic binary instrumentation.

If a check algorithm has negligible overhead, it can be applied not only in testing, but also after deployment at runtime. Then, bug detection is not restricted by the execution coverage of a test suite. Checks are possible at different levels: not only with source or binary instrumentation, but also in a managed runtime environment or with CPU acceleration (extra hardware). Examples for checks in a managed runtime are provided by the Java virtual machine. It features a bytecode verifier that checks, e.g., that data is always initialized and that references are type-safe. It further enforces access restrictions like '`private`' at runtime. Java also features the Java Virtual Machine Tool Interface, an API for the development of tools that need access to the state of the virtual machine. A modification of the Java virtual machine with integrated checks for data race conditions is evaluated in [EQT10]. An example for hardware-assisted runtime checks are the Intel Memory Protection Extensions (MPX), that can be used to detect memory access violations. MPX consists of extra bounds registers and instruction set extensions that operate on these registers. A proposal for hardware extensions to detect data race conditions is evaluated in [DWS^{+}12].

1.3 Constraint Solving and Automated Theorem Proving

Interesting software bugs are path-sensitive. For path-sensitive bug detection, a constraint solver is needed as logic backend. Typically, a Satisfiability Modulo Theories (SMT) solver is used. The satisfiability problem in Boolean logic (propositional satisfiability, SAT) is the problem of deciding whether there exists a valuation of variables, that satisfies a given Boolean formula (assignment of truth values so that the formula evaluates to true). The related problem of formula validity can also be decided with a SAT solver. A formula is valid, if it evaluates to true for all variable valuations. This means, that a formula is valid if its negation is not satisfiable. The SAT problem is known to be NP hard. However, many instances can be solved efficiently by equi-satisfiability transformations with the DPLL algorithm [DLL62]. The algorithm is a backtracking based search algorithm. Current solvers are based on the DPLL algorithm [DLL62] and conflict-driven clause learning [ZMMM01].

SMT is a generalization of SAT on the word level. It is a combination of background theories, expressed in first-order logic with equality. It features, for example, integer and real variables and arrays [dB11]. An SMT formula can contain predicates (Boolean valued functions) from different theories. An SMT solver can be implemented as combination of theory solvers with a SAT solver [NOT06], or the SMT problem can be reduced to SAT by ’bit-blasting’. Satisfiability is known to be undecidable in first-order logic, but is decidable for certain fragments of first-order logic. Of special practical interest are the theories of bit-vectors, floating-point numbers and arrays, that can model computer arithmetic with bit-precision. SMT can be seen as a form of the constraint satisfaction problem [Dec03]. An example state of the art SMT solver is Z3 [dB08]. Current solvers feature model generation functionality. If a formula is decided to be satisfiable, the solver can generate a variable valuation, i.e., an assignment of values to all variables, that satisfies the formula. The performance of SMT solvers is regularly compared at the Satisfiability Modulo Theories Competition[¶]. A standard interface to SMT solvers has been defined with the SMTlib [BST10]. The SMTlib is supported by current SMT solvers.

[¶]. www.smtcomp.org

1.4 Model Checking, Abstract Interpretation and Symbolic Execution

Symbolic analysis methods can be used to increase the input coverage to arbitrary input. Program input is treated as symbolic variables, and operations on variables yield logic equations. Symbolic methods can be used to automatically detect many types of common weaknesses independent of the availability of a test suite. Symbolic model checking, abstract interpretation and symbolic execution are closely related and partly overlapping approaches, that rely on a constraint solver (typically an SMT solver) as logic backend.

Infeasible perfect checker The perfect bug checker would be sound, complete, and have bounded runtime. Sound bug detection means that if a bug is reported, then it is indeed a bug (no false positives). Complete bug detection means that all bugs are found (no false negatives). Because the Halting problem is known to be undecidable in general [Tur37], the perfect checker can not exist. Any practical checker must drop at least one of the three properties.

Two types of checkers One type of bug checker is used by developers, in order to fix bugs as soon as they arise. It does not report too many false positives and preferably is sound. The other type of bug checker is used by quality assurance. It does not have too many false negatives and preferably is complete. Quality assurance can sort out false positives by hand, if the checker does not report too many. In the two cases, there is a different trade-off between the probability of missed bugs and the tolerable amount of time needed to sort out false positives.

Symbolic model checking Symbolic model checking typically translates the complete program into a formula, and evaluates properties of interest with a solver. Properties can be specified, e.g., in linear temporal logic (LTL) or computation tree logic (CTL) [BK08]. Model checking is typically applied as complete bug detection for verification, i.e., in order to prove a program bug-free. Verification normally stops at the first detected bug (when verification fails). Model checking faces a scaling problem, which is often called the combinatorial state explosion problem.

Abstract interpretation Abstract interpretation is a symbolic interpretation with automated abstraction, i.e., generalization of formulas. Abstractions overapproximate sets of bug-free satisfiable program paths, in order to avoid exploration of the complete set of satisfiable program paths. Abstract interpretation can be applied as complete bug detection for verification [CC77]. Scaling behaviour is improved by allowing false positives. Counterexample guided abstraction refinement (CEGAR) [CGL+03] is an iterative method that identifies and removes false positives.

Symbolic execution Symbolic execution automatically explores different paths through a program and collects path constraints [Kin76]. A path is satisfiable, if there exists at least one vector of program input, for that the program's execution takes the respective path. The set of a program's satisfiable paths is called the execution tree. The solver is used to decide path satisfiability and bug satisfiability for a given path constraint. The solver's model generation functionality can be used to generate an example input vector, that corresponds to the taken path and triggers a detected bug. Symbolic execution is typically applied as sound bug detection for testing. The scaling behaviour is improved by allowing false negatives [BS08]. Often, heuristic path exploration is used to quickly achieve a high degree of code coverage. The aim is typically to find as many bugs as possible in a given program, without false positives. Joint concrete / symbolic ('concolic') execution enables partial analysis of a program through consistent concretization [GKS05, SMA05]. Concretization leads to false negatives, but not false positives. An overview of available symbolic execution tools and applications is provided in chapter 6 on related work.

Overlap Symbolic execution can be applied both as a verification method [DE82, JNS11] and as a testing method. Some model checkers do not translate the complete program into one formula, but rather use static symbolic execution. In bounded model checking, loops are unrolled a bounded number of times before this underapproximation of the program's reachable states is translated into logic for incomplete bug detection. State interpolation [McM10, JNS11] in symbolic execution and model checking is a method for automated abstraction. CEGAR is seen as an implementation of model checking [CGL+03]. It has been noted in [JMNS12], that CEGAR starts abstract and through refinement becomes more concrete, while symbolic execution with state interpolation starts with concrete paths and abstracts them through interpolation. To enable verification of code functionality (rather than the absence of certain common weaknesses), source code has to be annotated adequately. In testing, typically unannotated code is checked for the presence of common weaknesses. Annotations in functionality verification reduce the context that has to be analyzed, and enable a compositional verification. Somewhere in between is extended static checking [DLNS98]: with a growing number of annotations (and an annotation checker), there is a smooth transition to functionality verification.

1.5 Evaluating Program Analyzers

Program analyzers are evaluated in practice by comparing the number of false positives, number of false negatives and analysis runtime on a sufficiently large test suite.

Two types of test suites Test suites for program analyzers are composed either of artificial test programs or of natural code. Natural code in this context are known bugs in (old versions of) open-source software, i.e., available bugs from the CVE list. The advantage of natural code is that the bugs are realistic, and that the test programs have realistic code size and complexity. An artificial test suite is typically composed of small test programs with intentional bugs. The advantage of an artificial test suite is the possibility of systematic coverage of bug types, syntax constructs and different context depths.

Juliet suite for C/C++ The Juliet suite [BB12] was developed by the US National Institute of Standards and the US National Security Agency's Center for Assured Software to evaluate and compare program analyzers. It is currently the most comprehensive artificial test suite for C/C++. In the current version 1.2 [Uni13] it covers 118 CWE types and 1617 flaw types. It consists of over 60000 small test programs with in sum over 8 million lines of code. It further covers 38 data and control flow variants for C. 'Baseline' bugs are systematically combined with the different flow variants, that cover the available programming language constructs. The maximum bug context depth spanned by a flow variant is five functions in five different source files. Each program in the suite contains both 'good' (without bug) and 'bad' (including a bug) cases in order to measure false positive and false negative detections.

1.6 Aims and Structure of the Text

Aims The work at hand deals with automated bug detection in C programs, with available source code. The aim is to detect as many common weaknesses as possible and as accurately as possible, within one analysis, with limited computational power. The detection is fully automated, i.e., without specification or annotations. Modification of source code is not necessary. Not all types of bugs can be detected fully automatically. An example of this is incorrect calculation. If a programmer means '+' but writes '*', this can not be detected as constraint violation (although it might lead to a detectable number overflow). The detection algorithms are suited for a tool for developers and therefore avoid false positives. Because these aims are at least partly conflicting, the approach of this thesis is a comparative investigation of algorithm variants of symbolic execution.

Structure The approach is a configurable analysis engine, that uses dynamic and selective symbolic execution on the source code level with interpolation based path merging.

Out of the known set of coverage criteria, some form a hierarchy with respect to implication. A test suite, that achieves for example MC/DC, does also achieve branch coverage. A test suite, that achieves path coverage (which is normally not possible because of an infinite execution tree), also achieves MC/DC. A coverage criterion can in general be achieved with different combinations of program paths. With respect to the hierarchy of coverage criteria, the respective sets of program paths can be chosen as subsets. The set of paths, that achieves MC/DC then contains the set of paths, that achieves branch coverage. Under this constraint, symbolic execution with higher code coverage (according to the hierarchy) detects at least an equal amount of bugs. It detects the bugs, that are reachable with the respective path set and are detectable as violations of automatically generated constraints. All bugs, that are detectable with symbolic execution, would be detected with path coverage.

Interpolation can be used to prune program paths as early as possible when path extensions can not contribute to achievement of a coverage criterion or to bug detection. Through interpolation, a chosen coverage criterion can normally be achieved with a smaller path set and correspondingly smaller computational effort. An interesting coverage criterion is 'error coverage', i.e., reaching all bugs that would be reachable with path coverage (considering only bugs, that can be detected as constraint violations). Through interpolation, error coverage is generally possible without path coverage, i.e., with a reduced path set. For error coverage it suffices to reach each bug on one path. It is not necessary to find all paths, that reach a bug. This work uses the approach of interpolation with unsat-cores and depth-first search from [JSV09]. In [JSV09], it is used for verification, i.e., to prove non-reachability of error locations. Interpolation with unsat-cores means omitting constraints from the path constraint. With this view, the approach can be generalized. Pruning paths based on live variable analysis [BCE08] then also falls into this framework, as it means omission of constraints for dead variables from the path constraint. The achieved coverage can be described as 'live context coverage'. If only unsat-cores of unsatisfiable branches are used for interpolation, then branch coverage is achieved. Considering the sets of constraints used for interpolation, then some sets again form a hierarchy as subsets. An interpolation hierarchy (constraint sets) corresponds to a hierarchy of coverage criteria and path sets. Omitting more constraints leads through more pruning of paths to a smaller path set and smaller coverage.

The implementation of symbolic execution with selectable interpolation is evaluated with test programs from the Juliet suite. Accurate bug detection in the Juliet suite requires modelling functions from the standard library and Posix (Portable Operating System Interface) library (e.g., symbolic execution does not follow system calls into the kernel). Because completely modelling these libraries would require an effort that goes beyond the scope of this thesis, the evaluation is limited to small subsets of Juliet's programs and standard library and Posix functions. The test programs are chosen to cover all data flow and control flow variants of the Juliet suite and to cover a variety of different common weakness types, namely buffer overflow, integer overflow, race condition, infinite loop, information exposure and dead code. The used test programs and modelled library functions are listed in the chapter on error detection.

The implementation follows the approach of concolic execution and uses a debugger machine interface for instrumentation. The implementation is a plug-in extension to the Eclipse C/C++ development tools (CDT) and uses CDT's code analysis framework and its debugger services framework. Z3 [dB08] is used as external process for constraint solving. Before static execution, a static taint analysis is done to detect (as over-approximation) decision nodes and potential error locations, that could be reached with symbolic variables. Based on this taint analysis, first breakpoints are set. Bugs that depend on symbolic input are detected during selective symbolic execution as violation of constraints, that are automatically generated during analysis. Constraints are generated at breakpoints. Breakpoints are set and removed path-sensitively. Any missing concrete values are queried from the debugger. The implementation is a symbolic tree-based interpreter. Constraints for bug conditions are generated at certain syntax constructs with symbolic parameters and are checked for satisfiability together with the path constraint. Bugs that only depend on concrete input are detected by compiler instrumentation. The implementation only partially models the C standard library, enough to detect errors in the test set. To detect more than the used few baseline bugs, an extended modeling of the standard library would be necessary.

The thesis is structured as follows: Chapter 2 describes the debugger-based selective symbolic execution on source level and its implementation in CDT. As much code as possible is executed concretely rather than interpreted symbolically. Chapter 3 illustrates methods for partial traversal of a program's execution tree. State interpolation and path merging and the algorithms for error coverage and branch coverage are illustrated. In Chapter 4, automated error detection with symbolic execution is presented in detail. Chapter 5 discusses the suitability of the described algorithms for further applications. Chapter 6 discusses related work. Results of the evaluation with the Juliet suite are discussed in Chapter 7.

Contributions

— A new algorithm for symbolic execution with branch coverage, based on interpolation of unsatisfiable branches with unsat-cores (Section 3.6.4.). The algorithm can be used, e.g., to detect dead code (Section 4.3.7). The algorithm can be applied along the same lines for MC/DC (Section 5.2).

— A new algorithm for detection of race conditions in multi-threaded code, based on symbolic execution with FIFO scheduling on one CPU core and instrumentation with happens-before analysis (Section 4.3.5). With this algorithm, multiple sequential errors can be detected on a path, because data races do not occur with this scheduling (but can often still be detected).

— Two small modifications of the interpolation algorithm from [JSV09]. The first (minor) modification is that potential error locations do not need to be annotated, but that the error conditions are generated from the syntax during analysis. This shows a difference between unsatisfiable branches and unsatisfiable error conditions. The second small modification allows further path pruning if the analysis is continued after detection of an error (precluding the detected error with an additional constraint). The modification consists in further weakening already computed interpolants. Constraints, that only stem from the (in previously analyzed contexts unsatisfiable) error condition are removed. This modification could identically also be applied, e.g., in [JMN13], where interpolation is combined with heuristic path exploration. The description can be found in Section 3.6.

— The qualitative description of a trade-off between computational effort and accuracy of bug detection. The trade-off stems from the algorithm variants of symbolic execution with selectable coverage through interpolation (from live context coverage over error coverage to branch coverage) and the description as coverage and interpolation hierarchy (Section 3.6.6).

— The quantitative description of the trade-off for the special case of a certain subset of the Juliet suite. The evaluation uses on the one hand the implementation-independent measures of false detection rates and path set sizes, and on the other hand the comparison of runtimes of the implementations within the same framework.

— A new algorithm for dynamic detection of infinite loops based on a modified autocorrelation with the Kronecker Delta function (Sections 4.3.3, 5.3).

— A new approach to symbolic execution of machine code. The standard approaches are either to write a dedicated symbolic execution engine for a certain processor architecture [GLM08, CARB12], or to lift instructions to an intermediate representation for symbolic execution [BJAS11]. The new approach consists in retargeting a symbolic execution engine based on an architecture description language (Section 5.7).

Chapter 2

Selective Symbolic Execution on the Source Code Level

2.1 Symbolic Execution as State Transition System

2.1.1 Single-Threaded Code

Formalization for a simplified language Symbolic execution can be seen as a state transition system [DE82, KKBC12]. This paragraph illustrates symbolic execution with a formalization from [KKBC12] shown in Algorithm 1, for a simplified language with only assignments, conditional goto statements, assertions and halt statements. Symbolic execution performs forward expression substitution starting from input variables, and can be described with the worklist algorithm [KKBC12]. A state is a triple (l, pc, s) with program location l, path condition pc and symbolic store s. The symbolic store s maps each variable to a concrete value or an expression (formula) over input variables. The initial program location is denoted l_0. States are kept in worklist w, a choice function $pickNext()$ chooses the next state to process from the worklist. The set of successor states is denoted S. The branch checker function $follow()$ decides whether to follow a branch (checks path satisfiability). In line 1, the worklist is initialized. $\lambda x.e$ denotes the function, that maps parameter x to expression e. In each loop iteration, one state is selected for processing from the worklist (line 3). In case of an assignment (line 6-8) $v := e$, the successor state has program location $succ(l)$. The expression e is evaluated by $\textbf{eval}(s, e)$ in the context of s, and v is mapped to the resulting (possibly symbolic) expression. Apart from literal expressions and identity expressions (variables), the simplified language contains only unary and binary expressions with arithmetic or logic operators. It does not contain function call expressions. If all operands in e are concrete, then $\textbf{eval}(s, e)$

evaluates to a concrete value. If at least one operand is symbolic, then **eval**(s, e) yields a formula: a symbolic operand in e is replaced by its formula. The new state is added to S. In case of a conditional goto statement (line 9-14), it is checked whether to take either branch (path satisfiability). If so, the branch condition is added to the successor state, and the successor state is added to S. In case of an assertion (line 15-20), the conjunction of path condition, symbolic store and negated assertion is checked for satisfiability; In case of a halt statement (line 21-23), the analyzed program is terminated and the path condition is output [KKBC12].

```
1  $w := \{(l_0, true, \lambda v.v)\}$ ;
2  while $(w \neq \emptyset)$ do
3      $(l, pc, s) := pickNext(w)$; $S := \emptyset$ ;
4      // Symbolically execute the next instruction
5      switch $instr(l)$ do
6          case $v := e$ do
7              // assignment
8              $S := \{(succ(l), pc, s[v \mapsto$ eval$(s, e)])\}$ ;
9          case $if(e)\ goto\ l'$ do
10             // conditional jump
11             if $follow(pc \wedge s \wedge e)$ then
12                 $S := \{(l', pc \wedge e, s)\}$ ;
13             if $follow(pc \wedge s \wedge \neg e)$ then
14                 $S := S \cup \{(succ(l), pc \wedge \neg e, s)\}$ ;
15         case $assert(e)$ do
16             // assertion
17             if $isSatisfiable(pc \wedge s \wedge \neg e)$ then
18                 abort ;
19             else
20                 $S := \{(succ(l), pc, s)\}$ ;
21         case $halt$ do
22             // program halt
23             print $pc$ ;
```

Algorithm 1: Symbolic execution, formalization from [KKBC12]

Control flow graphs On the source code level, the basis for analysis are Control Flow Graphs (CFG). A program consists of functions, that can be transformed into CFGs. The program's source files are parsed into abstract syntax trees (AST). The AST are then searched for function definitions. For each function definition, a CFG is constructed. There are different CFG node types:

Start node Each function begins with a start node.

Plain node A plane node has one parent and one child.

Decision node A decision node has several children, which are branch nodes.

Branch node A branch node has a label, which is normally 'then' or 'else'. If a branch node originates from a switch statement, it may have more than one sibling.

Connector node A connector node has several parents.

Exit node Each control flow graph has at least one exit node. Exit nodes normally originate from return statements.

A CFG node corresponds to a subtree of the file's AST, e.g., an expression statement or a declaration statement. Such a subtree may contain one or more function call expressions. Taking function calls into consideration, an extended CFG for the complete program can be constructed from the functions' CFGs by adding the following edges: In the extended CFG, connections are added from a CFG node that contains a function call to the called function's start node, and from the called function's exit nodes back to the calling node. A program location l is a CFG node. Successor locations $succ(l)$ are the children of l in the extended CFG.

2.1.2 Multi-Threaded Code

On a multi-core machine, different threads may run concurrently. Which combination of ready threads runs at which time is decided by the operating system scheduler. For the purpose of detecting software bugs including multi-threading bugs like data races or deadlocks, it is sufficient to consider only executions on one CPU core. An execution of a multi-threaded program on one CPU core is called a thread interleaving. Two executions of a multi-threaded program on one core may execute the same paths in all threads, but with a different thread interleaving (if the thread scheduling is different). For multi-threaded programs, the program state contains the program locations of all threads, that have been started, and one active thread (that will run next). The successor states are yielded by symbolic evaluation of the active thread's next CFG node and non-deterministic selection of the next active thread from the set of threads, that are ready to run. The program's multi-threaded execution tree is formed by the set of thread interleavings, that are satisfiable for any program input.

2.1.3 Traversing the Multi-Threaded Execution Tree

Within the framework of the worklist algorithm [NNH10], different tree traversal algorithms are possible:

Depth first search (DFS) A program path is followed till program end. Then, iteratively the last not taken branch on the path is further explored till program end. The worklist is a stack (last in, first out).

Breadth first search (BFS) The worklist is a queue (first in, first out).

Other Random or heuristic methods; a state scheduling algorithm is applied to select the next state to extend.

Completely traversing the multi-threaded execution tree obviously does not scale. For the purpose of accurate bug detection, full exploration is not necessary. The exploration methods with reduced complexity that are used in the work at hand (e.g., state interpolation and path merging), are presented in Chapter 3. Other methods like partial order reduction are discussed in Chapter 6 on related work.

2.2 Static Symbolic Execution using Eclipse CDT

Eclipse is a widely used open source IDE. Implementing symbolic execution as extension of Eclipse's C/C++ Development Tools (CDT) benefits from available framework functionality.

2.2.1 Extending CDT's Code Analysis Framework

CDT includes a C/C++ parser and a syntax tree visitor. The visitor class enables customized AST traversal according to the visitor pattern, which is described in [GHJV94]. CDT also includes a code analysis framework (Codan, [Las15]). Codan features a control flow graph builder and extensions of Eclipse's GUI for configuration, analysis start and error reporting. Codan currently (Eclipse release 4.5) does not feature any path-sensitive or context-sensitive analysis. CFG and AST are illustrated in Figure 2.1 for a simplified example function from the Juliet suite [BB12]. Important classes in CDT for CFG and AST are shown in Figure 2.2. There are different CFG node types for plain nodes, decision nodes, branch nodes, jump nodes etc. A CFG node (depending on the type) typically includes a reference to the corresponding AST subtree.

The work at hand is implemented as a plug-in extension for CDT, at Eclipse's extension point `org.eclipse.cdt.codan.core.model.IChecker`. It is assumed, that source code for the program under analysis is available.

2.2.2 Tree-Based Translation

Tree-based interpretation is a language implementation pattern [Par10]. Tree-based interpretation in this case means that for a CFG node the referenced AST subtree is interpreted. The AST subtree is traversed according to the visitor pattern. The traversal is typically bottom-up. The translation result can be a concrete value or a symbolic formula, depending on the AST subtree and the context. In case of a symbolic result, the AST subtree is translated into an SMTlib [BST10] logic equation, more

```
void simplified_memcpy_17_bad() {
  for(int j=0; j<1; j++) {
    charvoid cv_struct;
    cv_struct.y = (void *)SRC_STR;
    /* FLAW: Use the sizeof(cv_struct) which will overwrite the pointer y */
    memcpy(cv_struct.x, SRC_STR, sizeof(cv_struct));
  }
}
```

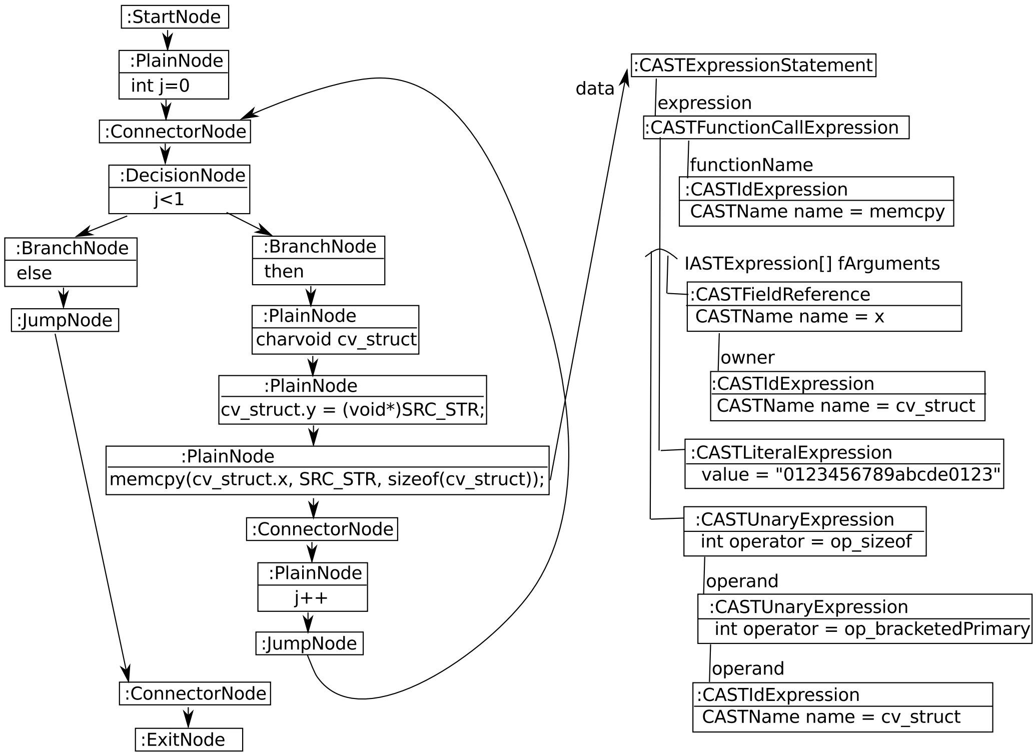


Figure 2.1 – Control flow graph and abstract syntax tree [Ibi13b]

specifically in the sublogic of closed quantifier-free formulas over the theory of bitvectors and bitvector arrays (QF_ABV). Bitvector lengths in the translation depend on the target architecture.

2.2.3 Deciding Path Satisfiability

The current Eclipse version does not feature an SMT solver as plug-in (only a SAT solver, [LP10]). Therefore, an SMT solver is connected as an external process. All evaluations in the work at hand use the Z3 SMT solver [dB08]. When a path reaches a branch node for whose parent's boolean decision a

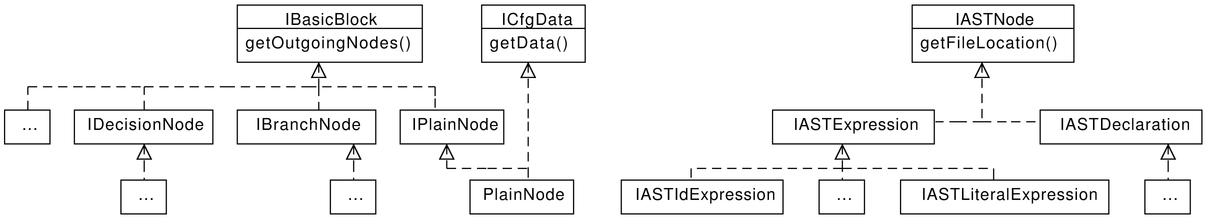


Figure 2.2 – Data structures for CFG and AST [Ibi15b]

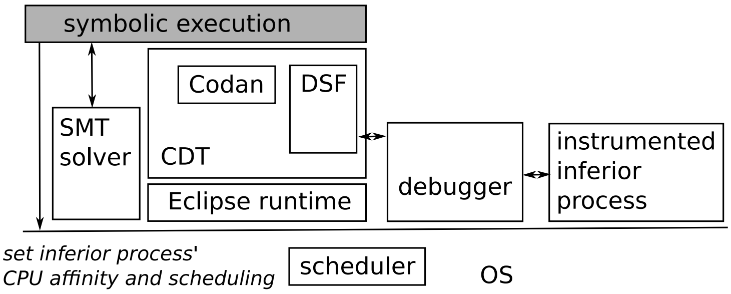

symbolic execution
SMT solver
Codan
DSF
CDT
Eclipse runtime
debugger
instrumented inferior process
set inferior process' CPU affinity and scheduling
scheduler
OS

Figure 2.3 – Main components

symbolic boolean variable was generated, then a constraint is added for the parent boolean decision. In case of a 'then' branch, the symbolic boolean is constrained to be true. The SMT solver is then used to decide whether the path constraint together with this branch constraint is satisfiable. For a satisfiable equation system, an SMT solver can generate a model, i.e., a variable valuation that satisfies the equation system. If the path is satisfiable, the solver's model generation functionality can be used to generate input, for which the program would take the current path in concrete execution. This can be used for automated test-case generation [CGP+06].

2.3 Offloading Work to Dynamic Analysis

Static symbolic execution can be combined with concrete execution into dynamic symbolic execution, which is also called concolic execution (mixed concrete / symbolic) [GKS05, SMA05]. The motivation is the following:

- Code execution is faster than interpretation, and especially faster than symbolic interpretation with a solver. Interpretation and translation into logic should therefore only be applied where necessary for the purpose of analysis.
- Dynamic symbolic execution offers the possibility for consistent concretization. Constraints are collected during concrete program execution, i.e., the full concrete program state always satisfies the collected path constraint. If certain program parts can not be handled with the solver, the respective symbolic variables can be concretized with a fall-back to their values in concrete execution. Concretization does not cause false positive error detections, but it does cause false negative detections (and misses some programs paths).

The work at hand implements a selective symbolic execution with debugger based dynamic instrumentation. The debugger is used to automatically drive execution into all program paths (measures for partial execution tree traversal are discussed in chapter 3). Selective symbolic execution allows to search for

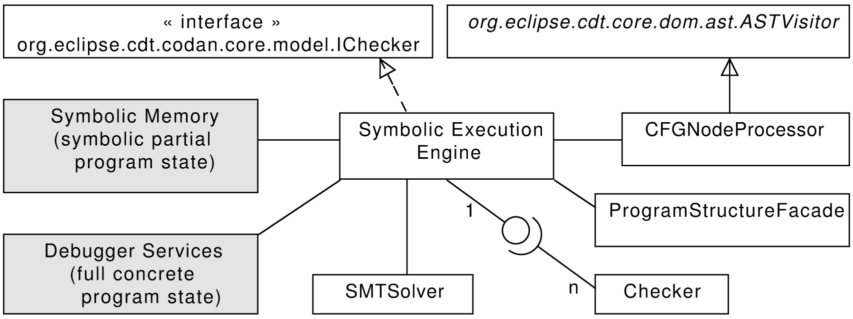

« interface »
org.eclipse.cdt.codan.core.model.IChecker
org.eclipse.cdt.core.dom.ast.ASTVisitor
Symbolic Memory (symbolic partial program state)
Symbolic Execution Engine
CFGNodeProcessor
ProgramStructureFacade
Debugger Services (full concrete program state)
SMTSolver
1
n
Checker

Figure 2.4 – Concolic execution [Ibi15b]

bugs symbolically only in selected source files and to not fork paths in other files, e.g., in libraries.

2.3.1 Selective Symbolic Execution with Tree-Based Interpretation

Execution of a program path changes between concrete execution in the debugger and symbolic interpretation. Debugger breakpoints are used to switch from concrete execution to symbolic interpretation. The debugger controls a full concrete program state. The interpreter contains the partial variable set, that is treated as symbolic, i.e., the values are logic formulas. C programs interact with their environment through functions from the C standard library (`libc`). The dynamic symbolic execution engine described in this section trace program input and determines a program path by forcing corresponding input.

Certain library functions are defined a-priori to have symbolic return variables. Correspondingly, initial breakpoints are inserted at call locations to the specified functions. The program argument vector is also treated as symbolic, i.e., breakpoints are set at locations where it is accessed.

Concrete variables may become symbolic, i.e., when they are assigned a formula. Then corresponding breakpoints at access to the new symbolic variable are set. Symbolic variables may become concrete, i.e., when they are assigned a concrete value. Then, the corresponding breakpoints are removed.

After reaching the program end, program input is automatically generated for the next path to explore. The solver and its model generation functionality are used to generate concrete input values, which are forced in the next program run. The input determines that the next program run will take a different branch, according to depth-first traversal of the execution tree.

An overview of the main components is shown in Figure 2.3. These comprise the Eclipse runtime with plug-ins for the C/C++ development tools (CDT). These contain a code analysis framework (Codan) and debugger services framework (DSF). The instrumented program under test is controlled through DSF using a debugger (here the GNU debugger `gdb`). The debug inferior process is scheduled by the operating system scheduler. Symbolic execution is implemented as Eclipse plug-in on top of CDT. Logic formulas are decided using the Z3 SMT solver [dB08].

An overview of the main classes is given in Figure 2.4. The symbolic execution engine performs tree-based interpretation [Par10] for program locations which use symbolic variables. The interpreter has a partial symbolic memory store which contains the symbolic variables (global memory and function space stack). For the rest, CDT's debugger services framework is used. For the detection of bugs that depend on symbolic input, the engine provides a checker interface. Through this interface, checker classes can register for triggers and query context information, which is necessary for the corresponding solver satisfiability checks.

The debugger stops at breakpoints or when the inferior process receives a signal. The symbolic execution engine then switches to symbolic interpretation and tries to resolve the respective CFG node and AST subtree. The source location (file and line number) can be obtained from CDT (CDT's `CSourceLocator`). In order to enable efficient CFG node resolution, a location map for the source files of interest is precomputed before analysis start. From the resolved CFG node a reference is then followed to its AST subtree, which is symbolically interpreted. Any needed concrete values are queried from the debugger.

Using CDT's debugger services framework Debuggers typically feature a machine interface (MI) to ease the development of graphical debugger frontends. CDT includes a debugger services framework (DSF) [PWCR08], which is an abstraction layer over debuggers' machine interfaces. DSF provides a set of asynchronous services. The main service interfaces are illustrated in Figure 2.5. They are mainly used to control dynamic execution with the debugger (`IMIRunControl`) and to insert breakpoints (`IBreakpoints`). The current program location and variables can be queried. This comprises local (`IStack`) and global variables (`IExpressions`).

Branches depending on symbolic input The debugger only breaks at a decision when the decision contains a symbolic variable (branch that depends on symbolic input). Possible branch targets (CFG branch nodes and their children) are obtained as children of the corresponding decision node. The debugger is commanded to step, and the taken branch is identified through the newly resolved CFG node. The branch constraint is formulated as symbolic formula. Branch constraints need to be remembered as part of the path constraint and to enable input generation for the next execution path. If there is already a breakpoint set for the source location after stepping, then this location is also symbolically interpreted. Otherwise the debugger is commanded to resume execution.

Implementation of read/write watchpoints

Concrete execution must be breaked at read and write accesses to symbolic variables. This means conceptually that a very large number of read/write watchpoints is needed. Software watchpoints would severely slow down debugger execution and in general are only available as write watchpoints, not read watchpoints [SPS11]. Hardware watchpoints can also not be used, since standard processors only support a handful of them. The implementation therefore uses normal software line breakpoints and determines the relevant locations using the available source code. To this end, a map of language bindings is precomputed before analysis. The map contains AST names with references to the corresponding source file locations. When a variable becomes symbolic, the corresponding breakpoints are inserted (through DSF's `IBreakpoints` interface, Figure 2.5). For accurate bug detection it is additionally necessary to trace pointer targets. Pointer assignments must be traced and considered because there may be pointers to a target when the target becomes symbolic (an example is presented in section 2.4).

Input generation

The parts of an execution path that are symbolically interpreted are called symbolic execution path in the following. To generate input for the next execution path, the symbolic execution path is backtracked to the last decision node. For any unvisited child branch nodes, satisfiability of the backtracked path constraint together with the respective branch constraint is checked using the solver. If the constraints are satisfiable, corresponding input values are generated using the solver's model generation functionality (SMTlib `get-model` command). If the constraints are not satisfiable, first the unvisited branch siblings are tested, then the symbolic execution path is further backtracked. Traversal of the symbolic execution tree is complete when further backtracking is not possible.

Function models and controlling program input

The implemented concolic execution consists of concrete program execution in the debugger with simultaneous partial symbolic interpretation on the same program path. Input enters the program under analysis through the argument vector and through function calls, especially through calls to the standard library. Functions can be configured as input functions. Then, a function model has to be provided. A function model serves two purposes. The first purpose is for symbolic interpretation. The model creates a symbolic variable for the return value and possibly also for function parameters that are written by the function. The second purpose is for the simultaneous concrete execution. The function model uses the debugger to enforce a certain concrete return value (and possibly set certain concrete values for function

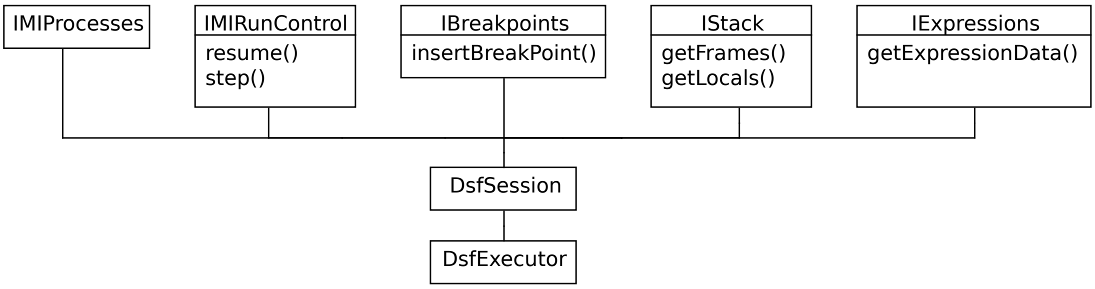


Figure 2.5 – Main interfaces of CDT's debugger service framework [Ibi15b].

parameters) so that the concrete execution takes the desired path.

Eclipse CDT and the symbolic execution plug-in including the function models are implemented in Java. A function model is implemented as a Java class. Debugger breakpoints are set on every C function for that a function model is registered. When the breakpoint is hit, the respective function model is executed. The function model can implement the following:

- Concrete values of return value and possibly other variables are enforced with debugger commands (`step into`, `return`, ...). Concrete value(s) on the first path are pre-configured in the model. On later program paths, the debugger commands are used to force input values generated by the SMT solver.
- Symbolic variables (for return value and possibly function parameters) are instanciated. Depending on the function that is being modelled, the return variables either remain unconstrained, or their symbolic values (formulas) are set in dependence on the (possibly symbolic) function's input parameters.

One example are input functions from the standard library (the modelled standard library functions are listed in Section 4.1.1). The `libc` contains several functions for which the debugger cannot step into or break inside. The reason is that these functions directly access the virtual dynamically shared object. Also preloading a wrapper library (with `LD_PRELOAD`) would not work in this case. To handle these cases, the program under test is compiled/linked for analysis with a linker wrap parameter (`ld --wrap`), which replaces symbols to redirect these calls into a wrapper library where the debugger can break. When for example malloc() is wrapped in this way, a call resolves to __wrap_malloc(). The system function becomes __real_malloc(). The implemented wrapper functions each contain only a function call to the real function, because the only purpose is to enable setting a breakpoint.

2.3.2 Thread Scheduling

Thread scheduling can be handled by the concrete part of concolic execution. One consideration might be to use the debugger to control the thread interleaving. But for this purpose, debuggers could only use single-stepping. The debug inferior process is scheduled by the operating system (OS) scheduler. Single-stepping would deteriorate the efficiency of selective symbolic execution. There are two options for selective symbolic execution of multi-threaded code with concrete scheduling:

User-space threads There are several user level thread libraries, that come with a user level scheduler. This scheduler can be controlled to execute the desired thread interleaving. This approach seems useful for very accurate detection of data races and deadlocks, in combination with partial order reduction. This approach is not used in the work at hand.

Operating system scheduler The operating system scheduler can be configured to apply deterministic scheduling on one core for the process under analysis. This yields the execution tree for the applied deterministic scheduling (without alternative thread interleavings). Due to the limitation to one fixed scheduling, in general not all bugs present in a program can be detected (false negative detections). Standard coverage criteria like for single-threaded code can then be applied. This option is implemented by the work at hand. Methods for the detection of races and deadlocks with deterministic scheduling are discussed in chapter 4.

2.4 Static Pre-Analysis

State pre-analysis can be used to reduce the number of locations that are interpreted, and to pre-compute information that is needed during symbolic execution. This static analysis is path insensitive. The implementation pre-computes:

— control flow graphs

— A location map for CFG node resolution. An AST node provides a reference to the corresponding source location. The computed map provides a lookup from source location to CFG node.

— initial break locations: program input functions and uses of the argument vector

— A binding map with reference to source locations. This is needed for the implementation of watchpoints. When a variable becomes symbolic, breakpoints are set on its usage (and re-definition).

— pointer analysis: the set of pointers that might (on any path) point to a symbolic target

— for automated error detection: potential bug locations depending on symbolic input, i.e., potential bug locations where a symbolic variable might be used. These locations must be symbolically interpreted in order to detect the bugs with a solver satisfiability check (Section 4.2) or, for state interpolation and path merging (Section 3.6), in order to compute an unsat-core.

Part of this information could also be computed in a lazy way instead of pre-computation. The analysis overapproximates the set of pointers that might on any program path point to a symbolic target. Breakpoints are set on the definition of such pointers, in order to trace the target. If a target becomes symbolic during symbolic execution, breakpoints are set on usage and definition of all pointers, that might point to this target. The static pre-analysis can be seen as taint analysis, or could be called 'maybe symbolic' analysis.

The analysis is implemented as a propagation of changes and uses the worklist algorithm [NNH10]. When the properties of a control flow node change, the change is propagated to its children (which are then added to the worklist). The analysis is not path-sensitive and does not need a constraint solver.

Figure 2.6 illustrates the need for pointer tracing with flow variant 32 of the Juliet suite ('data flow using two pointers to the same value within the same function' [Uni13]). The 'bad' function contains three variables `data` (declared in lines 2, 5 and 14). An initial breakpoint is set on the `fgets` function call in line 7. The second `data` variable becomes symbolic in line 9 due to the `atoi` library call, so that a breakpoint is set in line 12 (read access to this `data`). With an assignment through pointer dereference in line 12 the first `data` variable (from line 2) becomes symbolic. This would have been missed without tracing the pointer targets (here the pointer assignment in line 3). In line 12, also `data_ptr2` becomes symbolic, because it points to the now symbolic first `data`. Therefore also `data_ptr2` is watched, i.e., a breakpoint is set on line 14. In line 14, the third `data` variable becomes symbolic and is watched, so that the debugger breaks on lines 16 (where a constraint is collected) and 18. In line 18, solver bounds checks are triggered for the array subscript expression, and the buffer overflow is detected because the solver decides that the index expression might be larger than the buffer size. The detection of buffer overflow bugs depending on symbolic input is described in detail in section 4.3.1.

```
void CWE121_fgets_32_bad() {
  int data = −1;
  int ∗data_ptr1 = &data;
  int ∗data_ptr2 = &data;
  { int data = ∗data_ptr1;
    char input_buf[CHAR_ARRAY_SIZE] = ””;
    if (fgets(input_buf, CHAR_ARRAY_SIZE,
      stdin) != NULL)
    {  data = atoi(input_buf); }
    else
    { printLine(”fgets()␣failed.”); }
    ∗data_ptr1 = data;
  }
  { int data = ∗data_ptr2;
    int buffer[10] = { 0 };
    if (data >= 0) {
      // FLAW: possible buffer overflow:
      buffer[data] = 1;
      for(int i = 0; i < 10; i++)
      { printIntLine(buffer[i]); }
    }
    else
      { printLine(”ERROR:␣out␣of␣bounds”);}
  }
}
```

Figure 2.6 – Need for pointer target tracing; example from Juliet suite [BB12]

Chapter 3

Partial Execution Tree Traversal

It is neither practical nor necessary to traverse the complete combined scheduling and execution tree for multi-threaded software. This chapter deals with methods to reduce coverage in some form, with or without degradation of bug detection accuracy. All experiments in this chapter are run in an Eclipse version 4.5 (CDT 8.8.0) on a i7-4650U CPU, on 64-bit Linux kernel 3.16.0 with GNU debugger [SPS11] version 7.7.1.

3.1 Deterministic Concrete Scheduling

Exactly locating feasible data races is known to be NP hard [NM92]. For accurate detection of data races and deadlocks, different thread interleavings have to be explored. Typically, dynamic partial order reduction [FG05, AAJS14] is used for this purpose, discussed in chapter 6 on related work. The work at hand is content with faster approximate detection of data races and deadlocks (details in chapter 4 on error detection). For complexity reduction and acceleration, deterministic scheduling with the operating system scheduler is used. Compared to completely traversing the multi-threaded execution tree, the combinatorial explosion is drastically reduced. Symbolic execution is used to explore the execution tree of multi-threaded software for FIFO scheduling on a single CPU core.

Reproducible execution tree for multi-threaded programs Concolic execution needs a reproducible execution tree, also for multi-threaded software. This can be achieved by restricting the scheduling to one CPU core and to reproducible scheduling independent of outside parameters (independent of system load etc.). Then, the execution tree for this scheduling is yielded. The execution becomes reproducible: when restarted with the same program input, the identical thread interleaving is yielded. Here, FIFO scheduling on one CPU core is used. Bugs other than races (e.g., buffer overflows) can be found just like with symbolic execution of single-threaded code, and standard code coverage criteria like in single-threaded execution can be applied.

Reduced complexity at the expense of false negative bug detections The operating system scheduler is used in concolic execution for speed-up compared to scheduling in the symbolic interpreter. Only one thread interleaving is analyzed per program path. Due to the limitation to one fixed scheduling, in general not all bugs present in a program can be detected (false negative detections).

Prevents race conditions between threads The analysis of a program path should be able to continue after the detection of a potential data race, in order to achieve the desired code coverage. This means, that actual races should be avoided while still detecting them. Program behaviour without races is independent of scheduling. Data races between threads are prevented by using FIFO scheduling on one CPU core, because only one thread at a time is active, and it is not preempted by another thread (the thread executes until itself yields execution by calling the scheduler through a library call, e.g., from pthreads library). Potential data races can in many cases still be detected using happens-before, lockset or hybrid analysis. Signal handler race conditions can still occur.

Single-core FIFO scheduling on Linux The implementation currently runs on Linux, which supports different scheduling algorithms at the same time for different processes. Differing from the standard scheduler `SCHED_OTHER`, for the program under test the FIFO scheduler (`SCHED_FIFO`) is used. The CPU affinity is restricted to one CPU core. The corresponding Linux commands are `chrt` (to set the scheduling) and `taskset` (for CPU affinity).

3.2 Coverage Heuristics

A variety of path exploration heuristics have been proposed to achieve high branch coverage with limited computational resources. The approaches range from bounded exhaustive search over random search to informed search heuristics. In [GKS05, SMA05], one program path is analyzed at a time, with depth-first path enumeration. In [BS08], DFS is compared to a randomized approach and to a control-flow based heuristic. It is shown that DFS performs worst with respect to branch coverage on a set of example programs. The control-flow based heuristic searches for short satisfiable path extensions from a prefix of the current path to an uncovered program location. It randomly picks a branch with minimal sum of distance measure to uncovered code and number of previous branch flips during search. The distance to uncovered code is computed with Dijkstra's algorithm. It is concluded that the heuristic achieves ”significant branch coverage in practice” [BS08]. In [XTHS09] it is recognized that breadth-first search favors initial branches, while DFS favors final branches. A different heuristic function is proposed to determine a branch that is to be flipped for the next analysis path. The algorithms described in [CGP+06] and [CDE08] do not process one program path at a time, but rather use the worklist algorithm. Several active program states are kept in a worklist, and a heuristic function determines which state to extend next. At branch locations, a program state is forked into two. In [CGP+06], the state scheduling strategy switches between depth-first and best-first, where best-first extends the state whose code line has run the fewest number of times. In [CDE08], a random path strategy, a 'minimum distance to uncovered' strategy and a weighted random selection are used in a round-robin fashion. The weight function considers the minimum distance to an uncovered instruction, the call stack and whether the state recently covered any new code. Additionally there is a maximum number of instructions and a maximum number of time, that limit the extension of a single program state.

```
while (! (direction == exhausted) ) do
    if (direction == forward) then
        cfgnode = getNextNode();
        if (isProgramEnd(cfgnode)) then
            direction = backtrack;
            continue;
        interprete(cfgnode);
    else if (direction == backtrack) then
        while (!foundNewInputVec) do
            cfgnode = backtrackLastNode(path);
            if (isProgramStart(cfgnode)) then
                direction = exhausted;
                break;
            if (cfgnode instanceof DecisionNode) then
                if (hasUncoveredBranch(cfgnode)) then
                    boolean isSat = checkSat(path + newBranch);
                    if (isSat) then
                        InputVector newInput = getModel(path + openBranch);
                        foundNewInputVec = true;
        if (foundNewInputVec) then
            direction = forward;
            restart(newInput);
```

Algorithm 2: Simple branch coverage under-approximation

Simple Branch Coverage Under-Approximation

This subsection illustrates a simple path exploration strategy, that approximates branch coverage but in general does not achieve it. The algorithm is based on depth-first search, which would compute path

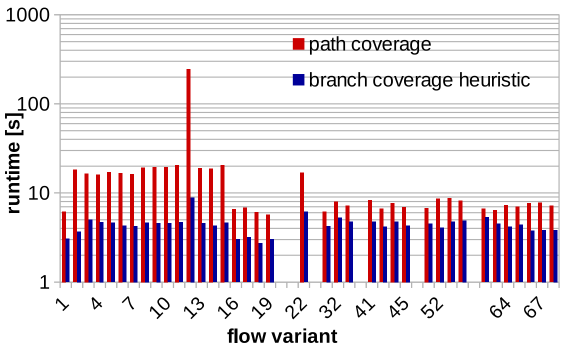


Figure 3.1 – Coverage heuristic, speed-up

coverage. The difference is in the backtracking mode.

In the following, a path's context means the set of all concrete and symbolic variables defined on the path. DFS backtracks a path to a branch node, that is satisfiable but not covered *in the current context*. The heuristic algorithm backtracks a path to a branch node, that is satisfiable in the current context and not yet covered *in any context*. The algorithm is depicted as pseudo-code in Algorithm 2.

The variable *direction* switches the algorithm mode between *forward* and *backtrack*, the algorithm terminates when *direction* takes the value *exhausted*. The variable *path* is a list of CFG nodes that corresponds to the program path being analyzed. The algorithm's forward symbolic execution mode spans lines 2-7. The variable *cfgnode* is the next control flow graph node (program location) to be interpreted and to be added to the *path*. The lines 9-22 belong to the backtracking mode. The boolean variable *foundNewInputVec* is set to true if backtracking finds a decision node with a yet uncovered child branch node, the conjunction of path condition with branch condition is satisfiable (solver call *checkSat()*, result in variable *isSat()*, and if the solver has generated the corresponding input vector (solver call *getModel()*). The input vector generated by the solver is *newInput*. Each entry in the vector contains the concrete program input for one input function. The algorithm partially traverses the execution tree, and with every switch from backtracking mode to forward symbolic execution mode, a new branch node in the program's CFGs is covered.

Complexity

In the following, a 'live branch node' denotes a branch node, that can be covered in some context (for arbitrary input there is a path on which the node is reachable). Conversely, a 'dead branch node' can not be covered in any context (unreachable). A 'symbolic branch node' denotes a branch node whose branch condition depends on at least one symbolic parameter. 'Live symbolic branch node' therefore means a branch node that is reachable on at least one path and whose condition depends on a symbolic parameter.

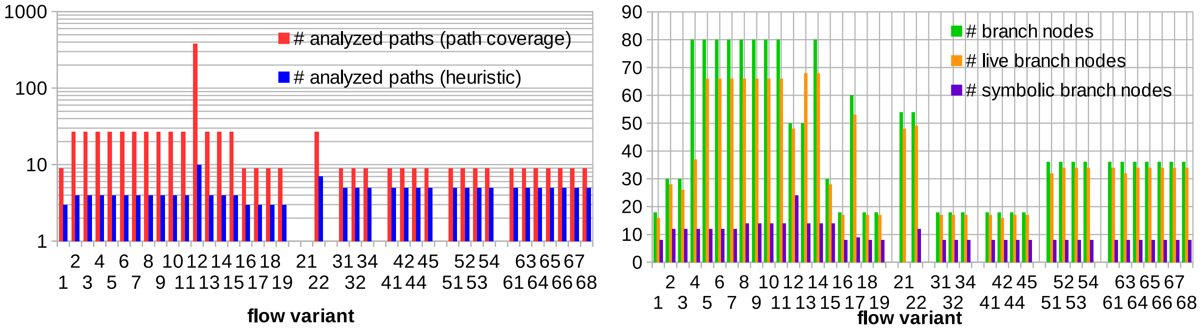


Figure 3.2 – Coverage heuristic, path reduction and branch node count

The algorithm only switches from backtracking mode to forward symbolic execution mode when there is a new input vector so that a new symbolic branch node is covered. This means, that the number of paths explored by the algorithm is bounded by the number of live symbolic branch nodes in the program. The number of branch nodes in a program grows linearly with program size. The number of paths, that the heuristic algorithm explores, therefore also grows at most linearly with program size.

Illustration

The algorithm is illustrated with 39 test programs from the Juliet suite [Uni13], that contain stack based buffer overflows with `fgets()`. The set of test programs used in this section contains just one of Juliet's many baseline bugs for buffer overflows, but it covers all of Juliet's data and control flow variants for C. The flow variants are listed in Table 4.1 on page 60. Aim of the experiment is first to validate accurate bug detection for all flow variants when running symbolic execution with path coverage (depth-first). Then, the heuristic algorithm is compared with respect to analysis speed-up and the number of missed bugs due to the reduced number of investigated paths. Further aim is to validate the complexity, i.e., the bound on the number of analyzed paths. The test set contains bugs that depend on symbolic input. The error condition is described as constraint violation in Section 4.2 on error detection during symbolic interpretation. The test programs are compiled for analysis. Address sanitization and coverage data collection (with `gcov`) is included in the binaries.

Speed-up Analysis runtimes for the test set are shown for path coverage and for the branch coverage heuristic in Figure 3.1. The figure uses a logarithmic scale. The horizontal axis shows the Juliet C flow variants (the flow variants in [Uni13] are not numbered consecutively in order to allow for later insertions). Only for flow 21 (control flow depending on static global variable) the runtimes are missing. This is because CDT's CFG builder misclassifies a branch node as dead, which leads to too few analyzed program paths and consequently to a false negative detection. The average speed-up on the test-set is about factor 4.

Number of analyzed program paths The number of analyzed paths for path coverage and for the branch coverage heuristic is shown in Figure 3.2 (on the left). It also uses a logarithmic scale. The high correlation of this figure with Figure 3.1 indicates that the reduction of the number of analyzed program paths is the main reason for the runtime speed-up. The average reduction of analyzed paths on the test-set is about factor 6. Coverage information is collected with `gcov` during symbolic execution. Figure 3.2 shows on the right side the number of branch nodes in the binaries (obtained from `gcov`). It also shows the number of live branch nodes. This number is obtained from `gcov` after running the tests with symbolic execution with path coverage. Additionally, the figure shows the number of symbolic branch nodes, i.e., the number of branches depending on symbolic input. This number is obtained from the symbolic execution engine in path coverage mode (a debugger breakpoint is only set for a decision if the decision depends on a symbolic variable). Figure 3.2 validates that the number of analyzed paths using the simple heuristic algorithm is indeed smaller than the number of symbolic branch nodes.

Bug detection accuracy The reduction of coverage is expected to cause false negative detections. Running the symbolic execution engine with path coverage correctly detects all bugs in the test set apart from flow 21, as explained above. Running it with the presented branch coverage heuristic algorithm only misses one additional bug, namely in flow 12. In this flow variant, the program takes different paths in dependence on random program input (`rand()` function). The engine is configured to enforce return value 0 for the `rand()` call on the first path. The return values on later paths are determined by the solver according to the presented path exploration algorithm.

3.3 Parallelisation

In order to achieve high code coverage in a limited time, parallelization of symbolic execution has been investigated. [SP10] presents a parallelized version of [APV07] and initially performs a breadth-first exploration of the symbolic execution tree up to a certain depth, and then runs multiple workers on the disjunct static partitions of the execution tree. [BUZC11] presents a parallelized version of `KLEE` [CDE08] with dynamic redistribution of work between workers.

For illustration, parallelization with multi-threading is applied here to (static) symbolic execution. Execution tree traversal is depth-first with backtracking of the symbolic program state. Different worker threads concurrently explore disjunct partitions of the execution tree, with dynamic re-distribution of work. Each worker of the parallelized engine keeps the symbolic program states along its current program path in memory, to allow for quick backtracking. Dynamic redistribution of work between workers is enabled by splitting a worker's partition of the execution tree at the partition's top decision node, where a partition is defined by the start path leading to its root control flow decision node. The child branches not taken by the current worker are returned as start paths for other workers. After a split, the partition start path is adjusted (prolonged by one branch). Analysis starts with one worker, who splits its work until the configured number of workers is busy. If a worker reaches an unsatisfiable branch or a satisfiable leaf of the execution tree, it backtracks and changes a path decision according to depth-first tree traversal. If backtracking reaches the end of the partition start path, the partition is exhausted.

Execution tree exploration and splitting into subtrees is illustrated in Figure 3.3, which shows part of the execution tree of the Juliet test program `CWE121_char_type_overrun_memcpy_12`. The tree shows only decision nodes and branch nodes; other CFG node types are not shown. The unexplored part of the (sub)tree is shaded. Red lines indicate the paths which have been explored by worker 1, green lines correspond to worker 2. Partition start paths are shown as solid lines, and a dashed line indicates the current position of a worker. Worker 1 splits its partition and generates a split path, which becomes start path for worker 2. After the partition split, worker 1's start path is prolonged by one branch. The execution tree is normally not generated during analysis, it is only traversed on-the-fly.

The worker threads share control flow graphs and abstract syntax trees. AST nodes are not thread-safe, so workers lock AST subtrees at the CFG node level (the AST subtree which is referenced by the currently interpreted CFG node). Each CDT project has an index, which is the persisted document object model (DOM). Access means (possibly blocking) I/O operations on a database stored in small files, so that workers acquire a read lock for accesses.

On the Juliet buffer overflow test cases with `fgets()` used in the last section, a 2x speed-up is achieved with three threads [Ibi13a].

3.4 Concretization

Variables become symbolic through assignment of a symbolic value. Concretization can always be used to simplify constraint systems, with a fall-back to concrete values in the debugger. Concretization leads to reduced coverage (due to control flow decisions that depend on symbolic input) and false negative bug detections, but also to an analysis speedup. Complexity (and bug detection accuracy) can be reduced

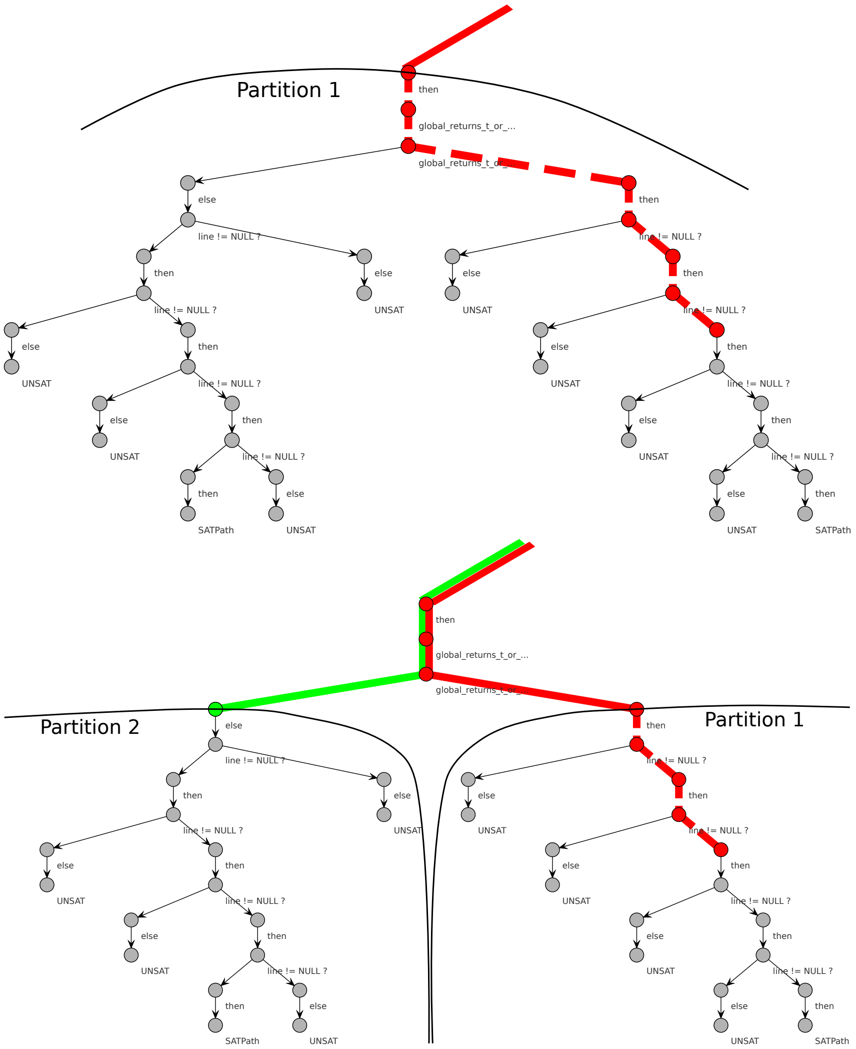


Figure 3.3 – Dynamic execution tree partitioning [Ibi13a]

by limiting the propagation of symbolic values with a propagation bound. Up to this point, potentially infinite propagation was described. A bound to level zero means, that if symbolic input is assigned to another variable, this variable is concretized. Bugs depending on symbolic input are then only detected, if the bug condition is *directly* reached with program input. A bound to level n means, that variables are concretized, that depend on program input indirectly over n steps.

3.5 Input Bounds

Program input like the argument vector or input from files and sockets can be restricted to a configurable limited size. This causes restrictions on the execution tree. Bounded input length can often avoid input-dependent infinite loops and cause a finite execution tree.

3.6 State Interpolation and Path Merging

A variable is dead if it is not read along any extension of the current path, otherwise (read on at least one path extension) it is live. A path's live context is yielded by removing all dead concrete and symbolic variables from the path's context. In many cases it is possible to prune a path from the execution tree without losing bug detection accuracy. Pruning a path means to stop analyzing it and to skip the subtree of all path extensions. This is possible if there is another path in the traversed part of the execution tree that (together with its subtree of path extensions) yields at least the same analysis results. In the following, pruning one path with reference to another one that is equivalent with respect to bug detection is denoted as merging one path into another.

In order to alleviate the path explosion problem, it is shown in [BCE08] that a live variable analysis can be applied so that program paths, that only differ in dead variables, can be merged without degrading bug detection accuracy. A more comprehensive path merging approach is described in [JSV09]. It is based on logic interpolation (Craig interpolation [Cra57]), i.e., on automated generalization of constraint formulas. The interpolation uses unsatisfiable cores (unsat-cores) and approximates weakest precondition computation.

The work at hand implements path merging with selectable interpolation. Section 3.6.6 describes a coverage and interpolation hierarchy, that includes the work described in [BCE08, JSV09] as special cases. Section 3.6.3 describes an algorithm that by way of comparison efficiently achieves error and branch coverage, section 3.6.4 describes one that achieves branch coverage (at the expense of missing some bugs). The work at hand only considers finite execution trees, i.e., programs with only a finite number of satisfiable program paths. Correctness of path merging for a finite execution tree is considered on pages 44ff. Completeness is neither claimed nor considered.

3.6.1 Path Merging Based on Live Variable Analysis

Constraints for dead variables are not generated for the path constraint. Paths can be merged, when their live context (including the stack) is identical. Merging in this way does not need a solver, it can be implemented as comparison (e.g., string comparison). This algorithm can be applied during breadth first path exploration with an over-approximation of live variables [BCE08]. It could also be applied during depth first exploration with more accurate liveness information from backtracking.

A path constraint P is the conjunction of constraints p_j collected on the path during symbolic execution:

$$P = p_1 \wedge p_2 \ldots \wedge p_N = \bigwedge_{j \in \mathcal{J}} p_j \quad , \quad \mathcal{J} = [1 \ldots N]$$

The N constraints p_j are numbered beginning from the program entry point. $\mathcal{J}$ is the set of natural numbers from 1 to N.

For a different path, the path constraint P' is:

$$P' = \bigwedge_{k \in \mathcal{K}} p'_k \quad , \quad \mathcal{K} = [1 \ldots N']$$

The path constraint P' consists of N' conjunctions, $\mathcal{K}$ is the set of numbers from 1 to N'. Removing constraints for dead variables from P means removing the p_j with $j \in \mathcal{J}_{\mathcal{D}}$, where $\mathcal{J}_{\mathcal{D}}$ indicates the set of indices of those constraints. This yields the live context:

$$\bigwedge_{j \in \mathcal{J} \setminus \mathcal{J}_D} p_j$$

The two paths P and P' are merged at the same program location $l_N = l'_{N'}$ under the following condition:

$$\exists N, N' \in \mathbb{N}. \quad \left(l_N = l'_{N'}\right) \quad \wedge \quad \Big(\bigwedge_{j \in \mathcal{J} \setminus \mathcal{J}_D} p_j \; = \bigwedge_{k \in \mathcal{K} \setminus \mathcal{K}_D} p'_{k'}\Big)$$

where $\mathcal{K}_D$ is the index set that indicates constraints for dead variables in P'.

An illustration with a control flow graph of a 'bad' function from Juliet is shown in Figure 3.4 (simplest flow variant, 'baseline' bug). There are 8 satisfiable paths through the CFG. On one path, there is a buffer overflow (indicated as red path). All operations are on stack variables, that die before function

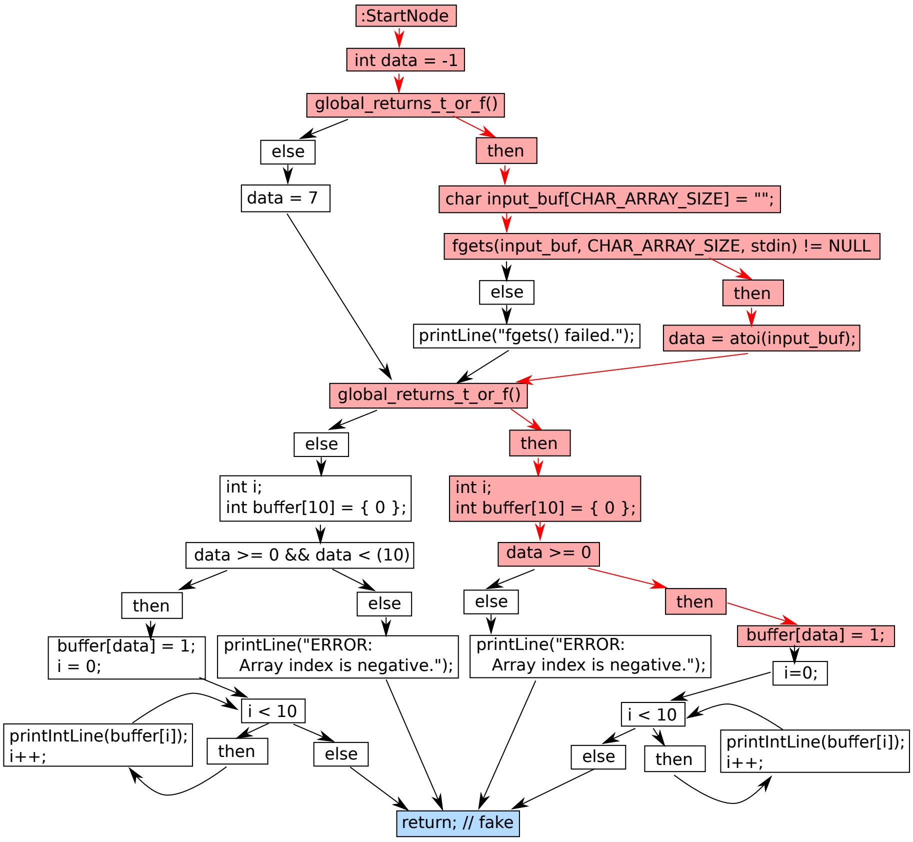


Figure 3.4 – There are 8 satisfiable paths through this CFG, on one there is a buffer overflow (red path). Path differences at function exit are only in dead variables, so path merging is possible [Ibi15c].

exit. Path differences at function exit are therefore only in dead variables. All paths can therefore be merged into one at function exit (the exit node is indicated in blue).

Another illustration with program execution tree is given in Figure 3.5. It shows on the top a reduced execution tree of a test program from the Juliet suite *, that only depicts decision nodes and branch nodes (other node types are omitted). There are four satisfiable paths through the program, that are indicated with a green 'SAT' label. The traversed execution tree with merging is shown on the bottom. Paths are merged at two locations, which is indicated with blue arrows. The figure shows that two sub-trees are skipped, which corresponds to folding the execution tree at two locations. The number of program paths, that are analyzed till program end, is reduced from four to one. Since path merging corresponds to folding the execution tree, one can expect an exponential effect on analysis run time.

3.6.2 Logic Interpolation and Path Merging with Unsatisfiable Cores

By leveraging the SMT solver, additionally some paths with different live contexts can be merged. The logic interpolation theorem [Cra57] states that ”in between” two formulas A and C with respect to

*. CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy_12

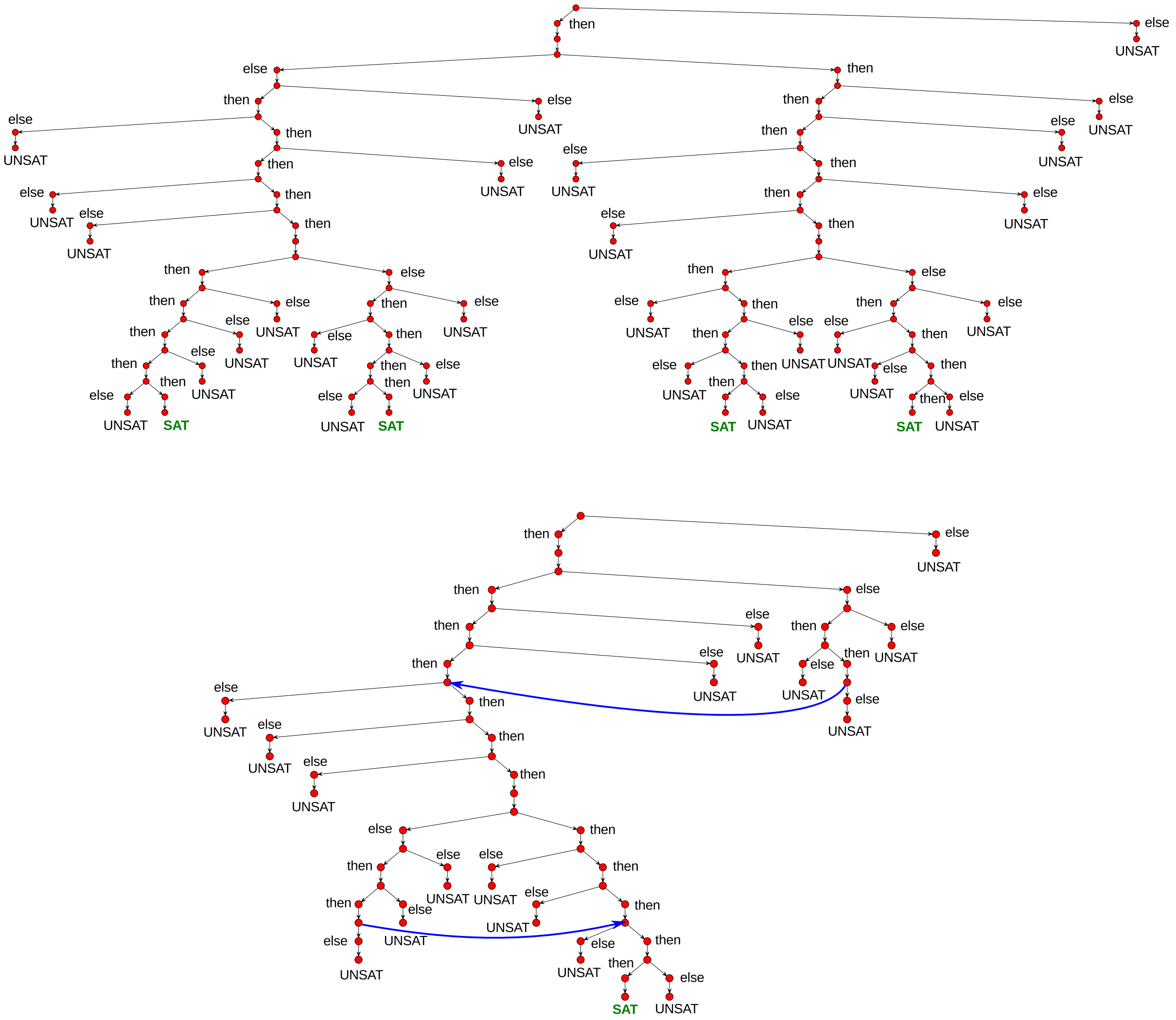


Figure 3.5 – Merging: folding the execution tree [Ibi15c]

implication there is another formula B.

$$\models \big(A \implies C\big) \quad \text{then} \quad \exists B. \Big(\models \big(A \implies B\big) \wedge \models \big(B \implies C\big)\Big)$$

Interpolation is an automatic way of generalizing (abstracting) formulas. The interpolant B is then generated. Some SMT solvers support interpolation. In symbolic execution, interpolation can be used to learn from unfeasible paths [JMNS12]. State interpolation in symbolic execution corresponds to conflict-driven clause learning in SAT solvers [McM10].

Unsatisfiable cores Given an unsatisfiable conjunction of formulas, an unsat-core is a subset of the formulas whose conjunction is still unsatisfiable. In a minimal unsat-core, every subset of formulas is satisfiable. Minimal unsat-cores are not unique. A minimum unsat-core contains the smallest possible number of formulas.

Unsat-core computation SAT and SMT solvers can compute unsat-cores. But for efficient path merging, it is advantageous to control the way in which unsat-cores are computed. Here, computation with serial constraint deletion [JSV09] is used. The algorithm goes once through the collected constraints from program start on. It sequentially tries whether the current constraint can be removed without rendering the conjunction satisfiable. If the conjunction stays unsatisfiable, the constraint is removed, and the algorithm proceeds with the next constraint.

3.6.3 Path Merging for Error and Branch Coverage

Motivation

In order to detect all detectable bugs, it is sufficient to detect each bug on one program path only. It is not necessary to detect each bug on all paths where this bug might be triggered. A program path can therefore be pruned from the execution tree (merged) if it is impossible to detect any new bugs on any extension of the path. This information can be gained from backtracking information about program paths, that were analyzed till program end. This implies a depth-first traversal of the program execution tree.

Algorithm

Interpolants are computed as unsat-cores for unsatisfiable paths and for unsatisfiable error conditions. The interpolants are backtracked as approximate weakest precondition (described later in more detail).

A merge formula denotes the conjunction of backtracked interpolants for unsatisfiable paths and unsatisfiable error conditions. A merge formula belongs to a program location to where the interpolants have been backtracked. A path can be pruned (merged) at a location, when it implies the location's merge formula. The locations, where path merging possibilities are checked (merge locations), are the branch nodes in control flow graphs (CFG). The key terms are:

Path constraint means the conjunction of the collected constraints along one program path.

Interpolant is the result of an interpolation, computed as an unsat-core.

Error condition means the context-free condition for occurrance of an error in a certain CFG node. Satisfiability of the conjunction of error condition and path constraint determines if the error may occur on a certain program path.

Interpolated error guard means an interpolant of an unsatisfiable conjunction of path constraint and error condition.

Merge formula Merge formulas are computed by the algorithm and assigned to CFG nodes. A path is merged if it reaches a CFG node where the path constraint implies the node's merge formula.

The algorithm overview is listed as pseudo-code in Algorithm 3. A debugger machine interface is used, and the functions $setBreaks()$, $continue()$ and $restart()$ perform the usual debugger actions of setting breakpoints, continueing execution and restarting execution. The static pre-analysis corresponds to line 1. Function $findInitialBreakLocation()$ determines the locations where initial breakpoints are set The depth-first selective symbolic execution corresponds to lines 4-37. It has a forward and a backtracking mode. The backtracking mode generates program input for the next path. The current path is backtracked to the last control flow decision that depends on symbolic input. If possible (satisfiable), the last decision is switched to obtain a new path.

Forward symbolic execution The forward symbolic execution mode corresponds to lines 5-15 in Algorithm 3. The debugger is run until it stops at a breakpoint. Function $isProgramEnd()$ (line 7) checks whether the end of the program is reached. If so, the algorithm switches to backtracking mode. Otherwise, function $getNodeLocation(Location loc)$ (line 14) resolves the CFG node for the location where the debugger breaked. This CFG node is then interpreted and translated into an SMT logic equation (function $interprete(cfgnode)$ in line 15). Values of concrete variables are queried from the debugger when needed. Functions from the standard library are modelled and/or wrapped, so that input can be traced and forced as desired using debugger commands (compare subsubsection "Function models and controlling program input" on page 19).

```
Set{Location} symlocs = findInitialBreakLocations();
debugger.setBreaks(symlocs);
direction = forward;
while (! (direction == exhausted) ) do
    if (direction == forward) then
        Location loc = debugger.continue();
        if (isProgramEnd(loc)) then
            direction = backtrack;
            continue;
        if (mergeLocs.contains(loc)) then
            if (cansubsume(loc)) then
                direction = backtrack;
                continue;
        cfgnode = getNode(loc);
        interprete(cfgnode);
    else if (direction == backtrack) then
        foundNewInputVec = false;
        while (!foundNewInputVec) do
            backtrackMergeFormula(cfgnode);
            if (cfgnode instanceof BranchNode) then
                setNewMergeLocation(cfgnode);
            cfgnode = backtrackLastNode(path);
            if (isProgramStart(cfgnode)) then
                direction = exhausted;
                break;
            if (cfgnode instanceof DecisionNode) then
                if (hasOpenBranch(cfgnode)) then
                    boolean isSat = checkSat(path + openBranch);
                    if (isSat) then
                        InputVector newInput = getModel(path + openBranch);
                        foundNewInputVec = true;
                    else
                        uc = getUnsatCore(path + openBranch);
                        setMergeFormula(openBranch, uc);
        if (foundNewInputVec) then
            direction = forward;
            debugger.restart();
```

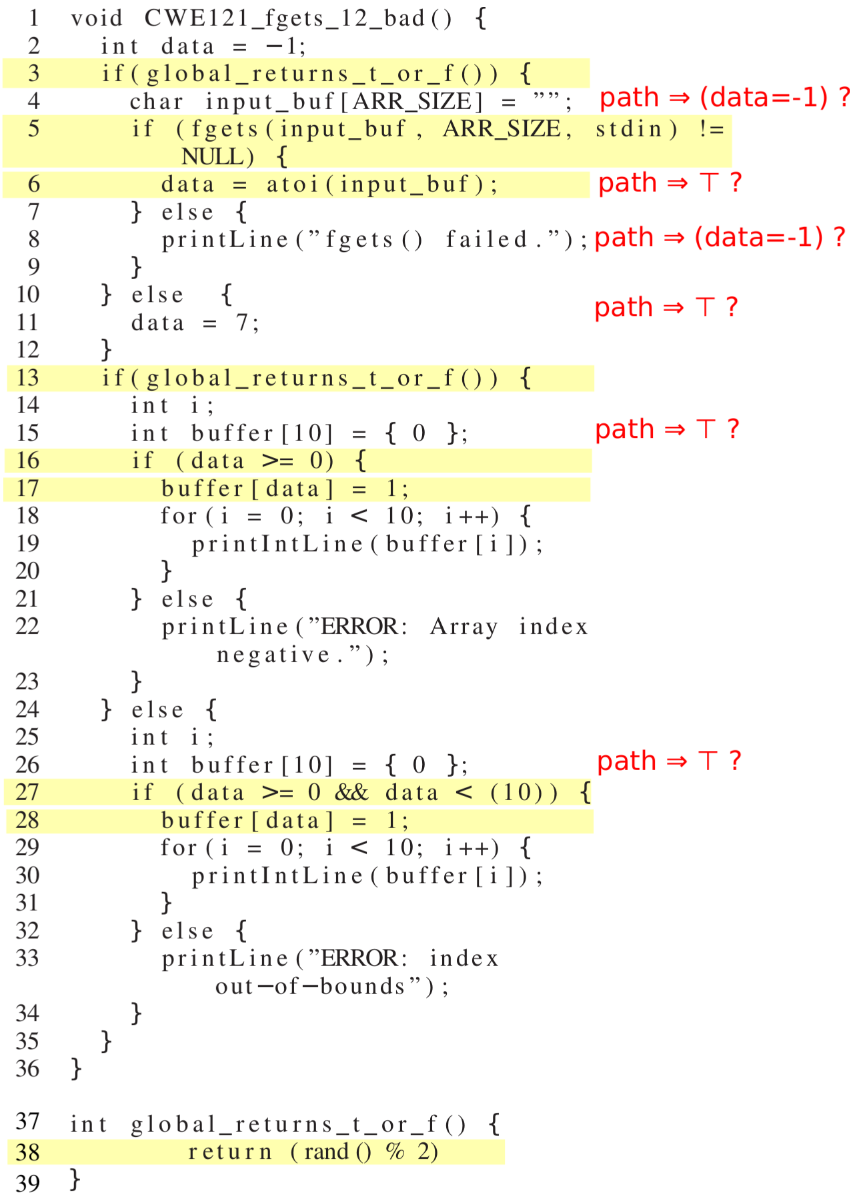
Algorithm 3: Dynamic symbolic execution with interpolation based path merging

Unsat-core interpolation for unsatisfiable bug conditions Another path can be merged if no new bug detection is possible along any of its extensions, i.e., when potential bug locations remain unsatisfiable for the new path. This is the case when the new path's path constraint implies the unsat-cores of the potential bug locations. Symbolic interpretation of a CFG node (function $interprete(cfgnode)$ in line 15) comprises error detection with the solver. At certain syntax elements, a corresponding checker is called. The checker then calls the solver to check satisfiability of the conjunction of path constraint and error condition. If the solver answers 'sat', the detected error is reported and the negated error condition is added to the path constraint. If the solver answers 'unsat', an unsat core interpolant is computed and added as interpolated error guard to the CFG node.

Updating for bug detections When a bug is detected, any new detection (re-detection) of the same bug (same type and location) becomes irrelevant. Therefore, any unsat-cores that were computed for this potential bug location before, can be deleted and the constraints removed from merge formulas (if these constraints do not also originate from other constituents of the merge formula). Unsat-cores are computed using the idea of serial constraint deletion from [JSV09]. A path constraint is a conjunction of a set of formulas. For each of these formulas it is checked in turn with the solver, whether the conjunction remains unsatisfiable if the formula is removed. The function is only kept if the conjunction would become satisfiable otherwise.

Path merging Breakpoints are set during backtracking for branch locations, for which a merge formula has been computed. The function $mergeLocs.contains(Location)$ (line 10) determines, whether a merge formula is available for the program location passed as parameter. When the current path constraint implies the merge formula, the path is pruned. Function $cansubsume()$ (line 11) checks the implication. The implication is valid if its negation is not satisfiable.

Backtracking The backtracking mode corresponds to lines 16-37 in Algorithm 3. It generates program input for the next path and generates merge formulas for program locations higher up in the program path (closer to program start) by backtracking merge formulas. Backtracking is only concerned with the partial symbolic program state along the current path, i.e., only with locations, that were symbolically interpreted. Function $backtrackLastNode(path)$ (line 22) removes the last node from the path. Function $isProgramStart(cfgnode)$ checks whether backtracking has reached the program's entry point. If so, the search is exhausted.



```
void CWE121_fgets_12_bad() {
  int data = −1;
  if(global_returns_t_or_f()) {
    char input_buf[ARR_SIZE] = "";
    if (fgets(input_buf, ARR_SIZE, stdin) !=
        NULL) {
      data = atoi(input_buf);
    } else {
      printLine("fgets() failed.");
    }
  } else {
    data = 7;
  }
  if(global_returns_t_or_f()) {
    int i;
    int buffer[10] = { 0 };
    if (data >= 0) {
      buffer[data] = 1;
      for(i = 0; i < 10; i++) {
        printIntLine(buffer[i]);
      }
    } else {
      printLine("ERROR: Array index
          negative.");
    }
  } else {
    int i;
    int buffer[10] = { 0 };
    if (data >= 0 && data < (10)) {
      buffer[data] = 1;
      for(i = 0; i < 10; i++) {
        printIntLine(buffer[i]);
      }
    } else {
      printLine("ERROR: index
          out−of−bounds");
    }
  }
}

int global_returns_t_or_f() {
        return (rand() % 2)
}
```

Figure 3.6 – Example function for interpolation based path merging [Ibi16a]

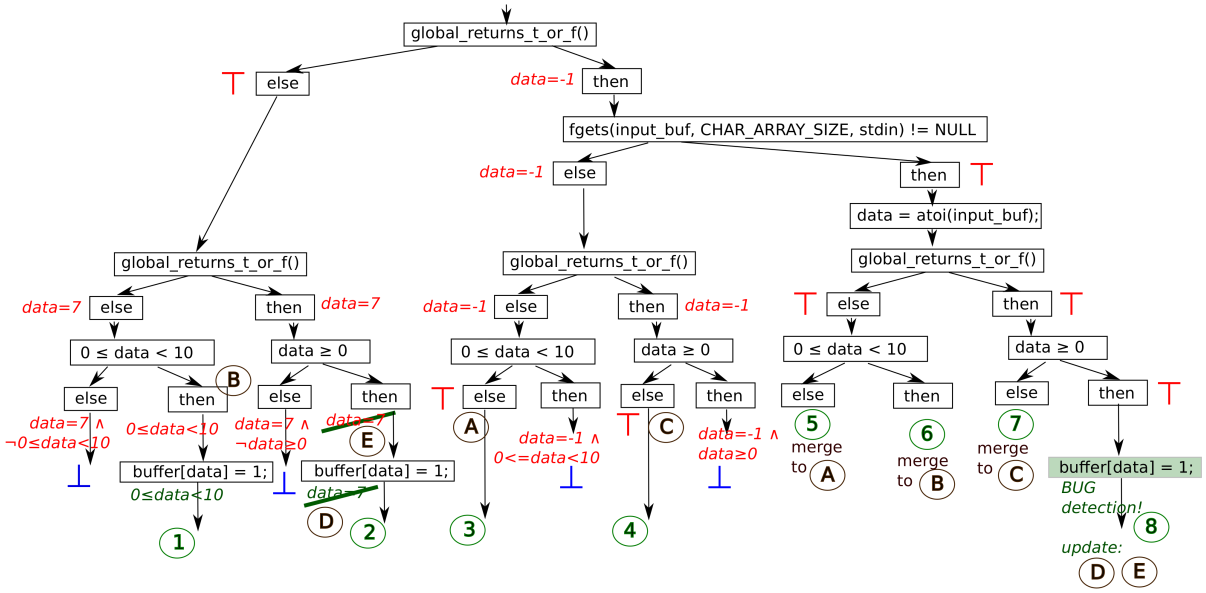


Figure 3.7 – Algorithm progress [Ibi16a]

Input generation The symbolic program state is backtracked to the last decision node. Function $hasOpenBranch(cfgnode)$ (line 27) checks whether the decision node has a child branch node that has not been visited in the path's context. For child branches, that were not yet covered in the context of the current (backtracked) path, it is checked with the solver whether or not this path extension is satisfiable. Function $checkSat()$ (line 28) wraps the SMT solver call to check satisfiability. If it is satisfiable, the solver's model generation functionality is used to generate corresponding program input values for the next path. Function $getModel()$ (line 30) wraps the SMT solver call for model generation (input generation for the next path). If not, an unsat-core is computed and set as merge formula, and the symbolic program state is further backtracked.

Unsat-core interpolation for unsatisfiable paths Because unsatisfiable paths are not further explored, any potential error locations after the unsatisfiable branch are not evaluated in this context. Another path can therefore only be merged as long as unsatisfiable branches remain unsatisfiable. This means, that an unsat-core for an unsatisfiable path could also be seen as an interpolated error guard. Function $getUnsatCore()$ (line 33) performs interpolation by removing as many constraints as possible while keeping the result unsatisfiable and returns the computed unsat core. Function $setMergeFormula()$ (line 34) sets the computed interpolant as merge formula for the branch node whose branch is unsatisfiable in this context.

Backtracking merge formulas A merge formula is generated when the program end is reached (at exit node of main() function). A merge formula is also computed for an unsatisfiable path as unsat-core. Interpolated error guards at the locations of unsatisfiable bugs on the path are computed as unsat-cores in forward symbolic execution. Backtracking also backtracks merge formulas. The approach is approximate weakest precondition computation [JSV09, JMNS12]. While forward symbolic execution computes strongest postconditions, this could be seen as ’reverse symbolic execution’. When a node’s child is backtracked, then any constraints which were generated in this child (as symbolic interpretation) are removed from the node’s merge formula. Because constraints are removed, backtracking means a generalization of merge formulas. Decision nodes are the only control flow node type that has more than one child node, i.e., several branch nodes. The children’s contributions to a decision node’s merge formula are determined during backtracking as the conjunction of the children’s merge formulas. The reason is that path merging requires that no new bug detection becomes possible on *any* extension of the current path. When backtracking reaches a branch node, a breakpoint is set and associated with the merge formula. Function $backtrackMergeFormula(cfgnode)$ (line 19) backtracks the computed merge formula over one CFG node (the node is the call parameter). Function $setNewMergeLocation(cfgnode)$ (line 21) sets the merge formula for the program location that is given as parameter.

Implementation Interpolation based path merging means that more breakpoints are set than without merging. On a path that can not be merged, more locations are symbolically interpreted than without merging. The implication check for merging further requires variable projections as described in the following. Single assignments are used in the translation to logic to avoid destructive updates, i.e., single assignment names are used for variables in the logic equations. Because a merge formula was computed on a different path and the translation into logic uses single assignment names, a subset of variable names in both formulas (merge formula and path constraint) has to be substituted. These variable names are the last single assignment versions in both formulas of variables whose definitions reach the merge location. These variables are projected (substituted) to the corresponding syntax tree names (names in the source code). Because branch nodes are not necessarily explicit in the source node, the merge locations are not exactly branch nodes, but rather the next following program location where a debugger breakpoint can be set. This is the following expression or declaration with initializer. Merge location examples are given in Figure 3.6.

Example

To illustrate the algorithm, it is applied to the example function shown in Figure 3.6. The example is a ’bad’ function from the Juliet suite [BB12], which contains a path-sensitive buffer overflow in line

17. The standard library functions `fgets()`, `atoi()` and `rand()` are treated as giving arbitrary (unconstrained) symbolic input. These functions are called in lines 5, 6 and 38. Static pre-analysis additionally yields breakpoints for lines 3, 13, 16 and 27 as symbolic input dependent control flow decisions, and for lines 17 and 28 as symbolic input dependent potential error locations. Together, these lines are indicated as shaded in the figure. Breakpoints are set on these lines, so that the debugger stops there and they are symbolically interpreted. The remaining not shaded locations are always just executed concretely, the debugger is not stopped for them. For the example, we assume that the function is called directly before program end, i.e., there are no backtracked formulas from other functions called later.

Algorithm progress is illustrated in Figure 3.7. The explored satisfiable program paths are marked with green numbers 1-8 in exploration order. Unsat-cores for unsatisfiable bugs are shown as green formulas (on path 1 and 2). Unsatisfiable branches are marked with a blue 'false' symbol ($\perp$, four times). Interpolants are computed as unsat-cores. Backtracked merge formulas are shown as red formulas next to the respective control flow nodes. Merge locations are the branch nodes, i.e., 'then' and 'else'. The merge formulas are shown in red next to them. The 'true' symbol ($\top$) indicates an empty backtracked merge formula (7 times). Path merges occur during the exploration of paths 5, 6, and 7. The respective merge targets are marked 'A' to 'C'. The bug is detected on path 8. This potential bug was unsatisfiable on path 2. The respective computed (backtracked) unsat-core is removed, because any re-detection of this bug on another path would be irrelevant. By removing constraints, more path merging can become possible in general. The updated tree nodes are marked 'D' and 'E', the removed constraints are crossed out in green.

Analysis results are also illustrated in Figure 3.6: merge locations and the corresponding merge formulas (implications) are indicated in red on the right side of the figure. For any further call of this function in the program under analysis, all paths can be merged at latest in lines 15 or 26 (because the respective implications are always valid).

Correctness

The path merging algorithm for error and branch coverage is correct, if the same set of errors is detected and the same branch coverage is reached compared to symbolic execution without merging. This is true if only paths are merged (pruned), that can not cover previously uncovered code and can not detect previously undetected errors. The proof idea is backwards structural induction, starting from program end. The description uses different types of control flow graph nodes like exit node and decision node. CFG node types are described on page 12. The path exploration order of the algorithm is depth-first. It is also shown as an intermediate result that merge formulas by way of construction are (like path constraints) conjunctions of constraints, where each of the constraints is present in at least one path constraint (selected by serial constraint elimination, unsat-core computation).

Interpolation at program end When symbolic execution reaches an exit node of the main() function (i.e., program end), the merge formula for this location is set to:

$$M = \perp$$

This can be seen as an empty conjunction of constraints (zero constraints, unconstrained). This means that every other path reaching this node can be merged, because true is implied by all path constraints. This merging is correct, because it is impossible that new code is covered or that a new bug is found after an exit node of the main() function.

Interpolation of unsatisfiable paths Any path constraint P is a finite conjunction of the constraints collected along the path:

$$P = p_1 \wedge p_2 \ldots \wedge p_N = \bigwedge_{j \in \mathcal{J}} p_j \quad , \quad \mathcal{J} = [1 \ldots N]$$

An unsatisfiable path is interpolated at the first branch node where the path constraint becomes unsatisfiable. That branch constraint is denoted B.

$$(P \wedge B) \text{ unsat}$$

means that the following is valid:

$$\neg(P \wedge B)$$

Interpolant I is computed as unsat-core of $P \wedge B$ by serial constraint elimination from P:

$$I = \bigwedge_{j \in \mathcal{K}} p_i \quad , \quad \mathcal{K} \subseteq \mathcal{J}$$

The index set $\mathcal{K}$ is a subset of the index set $\mathcal{J}$, i.e., I contains a subset of the constraints that P contains. The merge formula M is:

$$M = I$$

Another path P' is merged if

$$P' \implies M$$

This means that:

$$P' \wedge B \implies I \wedge B$$

From

$$(I \wedge B) \text{ unsat}$$

it follows that

$$(P' \wedge B) \text{ unsat}$$

i.e., the path P' cannot take branch B. Therefore merging P' into M at this location is correct.

Backtrack merge formula over one control flow node to non-decision node Consider a CFG node L_1 with child node location L_2. Assume that the merge formula at the child node is M_2 and that it is correct. Assume that the merge formula M_2 is a conjunction of constraints that occur in path constraints.

There are four possibilities:

1. M_2 does not contain the constraint C_1 that was generated by symbolic interpretation of L_1, and M_1 is not a location with a potential error that is not yet detected. The merge formula M_1 at L_1 is the same as the one at L_2.

   $$M_1 = M_2$$

   Then the following holds true:

   $$(P' \implies M_1) \qquad \implies \qquad \{\, (P' \wedge C_1) \implies M_2 \,\}$$

   A path P' that is merged at L_1 would have been merged at L_2 anyway.

2. M_2 contains the constraint C_1 that was generated by symbolic interpretation of L_1, and M_1 is not a location with a potential error that is not yet detected. The constraint C_1 is removed from M_2 to yield M_1.

   $$M_2 = M_1 \wedge C_1$$

   $$M_1 = M_2 \setminus C_1$$

Then the following holds true:

$$(P' \implies M_1) \qquad \implies \qquad \{\, (P' \wedge C_1) \implies (M_1 \wedge C_1) \,\}$$

A path P' that is merged at L_1 would have been merged at L_2 anyway.

3. M_2 does not contain C_1, but L_1 is a potential error location and the error condition $E1$ is unsatisfiable under the current path constraint. The interpolated error guard is G_1, computed by serial constraint elimination from the path constraint. Therefore G_1 is a conjunction of constraints. The merge formula M_1 at L_1 becomes:

   $$M_1 = M_2 \wedge G_1$$

   Then the following holds true:

   $$\{\, P' \implies (M_2 \wedge G_1) \,\} \qquad \implies \qquad \{\, (P' \wedge C_1) \implies M_2 \,\}$$

   A path P' that is merged at L_1 would have been merged at L_2 anyway, and the potential error at L_1 is not satisfiable on P'.

4. M_2 contains C_1, and L_1 is a potential error location where the error condition $E1$ is unsatisfiable under the current path constraint. The interpolated error guard is G_1, computed by serial constraint elimination from the path constraint. Therefore G_1 is a conjunction of constraints.

   $$M_1 = (M_2 \setminus C_1) \wedge G_1$$

   Then the following holds true:

   $$\{\, P' \implies ((M_2 \setminus C_1) \wedge G_1) \,\} \qquad \implies \qquad \{\, (P' \wedge C_1) \implies (M_2 \wedge C_1) \,\}$$

   A path P' that is merged at L_1 would have been merged at L_2 anyway, and the potential error at L_1 is not satisfiable on P'.

In all cases, M_1 is a conjunction of constraints that occur in path constraints (the ones that were generated in L_1 are dropped from M_2 and any interpolated error guard is added as conjunction. The assumed correctness of merging at L_2 into M_2 implies correctness of merging at L_1 into M_1.

Backtrack merge formulas from control flow branch nodes to decision node This case is illustrated in Figure 3.8. Location L_3 is a decision node and has the two branch nodes L_1 and L_2 as children. The branch condition from L_3 to L_1 is denoted B. Merge formula at L_1 is M_1, the merge formula at L_2 is M_2. The merge formula at L_3 becomes:

$$M_3 = (M_1 \setminus B) \ \wedge \ (M_2 \setminus \neg B)$$

Given that M_1 and M_2 are conjunctions of constraints, then M_3 also is a conjunction of constraints. A new path P' is merged into M_3 if

$$P' \implies M_3$$

This means that P' implies both $M_1 \setminus B$ and $M_2 \setminus \neg B$:

$$(P' \implies (M_1 \setminus B)) \ \wedge \ (P' \implies (M_2 \setminus \neg B))$$

This yields both

$$(P' \wedge B) \implies M_1$$

and

$$(P' \wedge \neg B) \implies M_2$$

This means, that if $P' \wedge B$ is satisfiable, the path would have been merged at L_1 anyway. And if $P' \wedge \neg B$ is satisfiable, it would have been merged at L_2 anyway. Therefore merging P' at L_3 into M_3 is correct. If the condition node L_3 is a potential error location of a yet undetected error that is unsatisfiable on path P, and the interpolated error guard is G, then merge formula M_3 becomes:

$$M_3 = (M_1 \setminus B) \ \wedge \ (M_2 \setminus \neg B) \ \wedge \ G$$

The correctness argumentation is identical, because:

$$((M_1 \setminus B) \ \wedge \ (M_2 \setminus \neg B) \ \wedge \ G) \implies ((M_1 \setminus B) \ \wedge \ (M_2 \setminus \neg B))$$

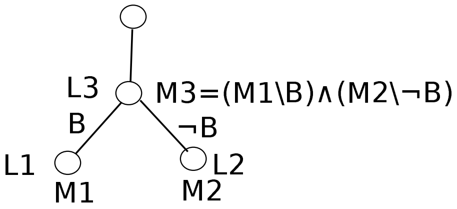


Figure 3.8 – Backtracking merge formulas

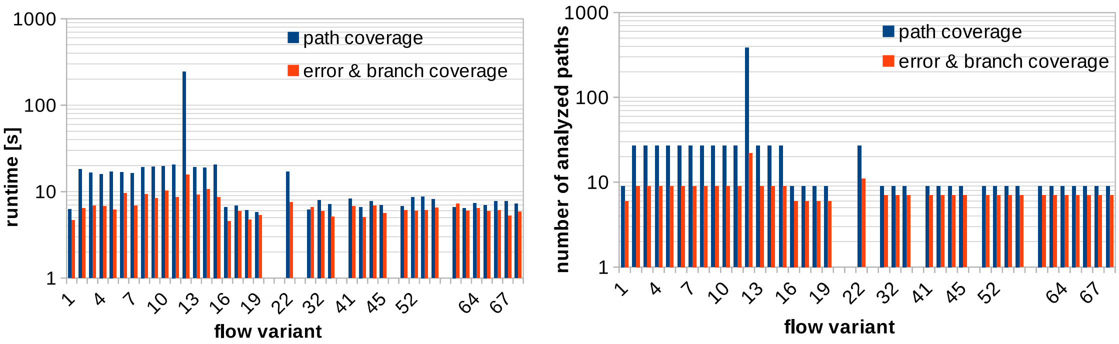


Figure 3.9 – Run-times with and without path merging [Ibi16a]

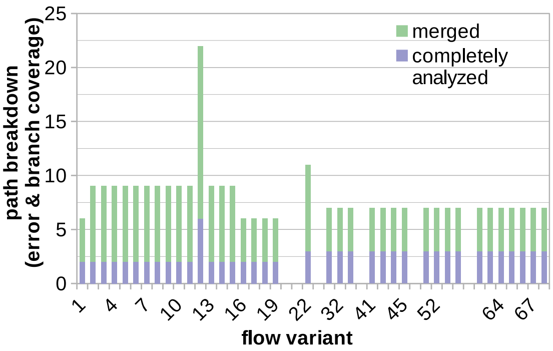


Figure 3.10 – Breakdown of the number of analyzed paths with merging [Ibi16a]

Experiments

The implementation is evaluated with the Juliet buffer overflow test cases with `fgets()` from the last section. The error condition is described as constraint violation in Section 4.2 on error detection during symbolic interpretation. Aim of the experiments is to validate an analysis speed-up with path merging for error and branch coverage without loss of detection accuracy compared to path coverage (while path merging does require extra computation of interpolants, merge formulas and implication checks, an overall speed-up is expected). Another aim is to validate the expected main reason for speed-up, i.e., a reduction of the number of analyzed program paths (merging skips a subtree of the execution tree). The results are shown in Figures 3.9 and 3.10. As expected, both the presented algorithm and the path coverage algorithm accurately detect the contained errors for all flow variants except for flow variant 21 (CFG builder exception).

Speedup with merging The measured run-times both with and without path merging are shown in Figure 3.9. The figure's time scale is logarithmic. Despite the additional analyses for path merging, there is a clear speed-up for all test cases. The biggest speed-up is achieved for flow variant 12, which also contains the largest number of satisfiable program paths.

Reduction in the number of analyzed paths Figure 3.9 on the right also shows the number of analyzed paths with path coverage on the one hand and with error and branch coverage on the other. There is a clear reduction in the number of analyzed paths for all test programs. Merging prunes a subtree, which in general splits into more than one satisfiable program path. The figure shows a strong correlation with Figure 3.9 on the left, so that the reduction in the number of analyzed paths can be seen as the main reason for the speed-up.

Proportion of merged paths Figure 3.10 shows a breakdown of the analyzed paths for merging only (error and branch coverage). The analyzed paths are distinguished into paths, that are completely analyzed until program end, and others, that are merged at some point. The figure shows that for all test cases the majority of analyzed paths is merged at some point, which is an additional reason for speed-up and explains another part of it (with the reduction of the analyzed lengths of the paths that are merged).

3.6.4 Path Merging for Branch Coverage

Path merging for branch coverage is a special case of the presented algorithm for error and branch coverage. Only unsat-cores for unsatisfiable branches are computed, unsat-cores for unsatisfiable potential errors are not computed. A path is then pruned as early as it becomes clear that no extension can take any previously unsatisfiable branch.

3.6.5 Path Merging for Modified Condition / Decision Coverage

For MC/DC, the described branch coverage algorithm can be applied with a minor adaptation. The adaptation is to compute unsat-cores for unsatisfiable boolean constraints over individual decision predicates. Conceptually this corresponds to transforming the source code by splitting decisions into a concatenation of simple decisions, and then applying the branch coverage algorithm.

3.6.6 A Coverage and Interpolation Hierarchy

A path constraint is the conjunction of constraints collected along the path. Interpolation is considered in this thesis as removal of constraints from the path constraint. Code coverage can be changed by varying interpolation. With 'more' interpolation it is meant here to remove more constraints from the path constraint. In this section it is assumed that both the algorithms for branch coverage and for error & branch coverage compute interpolants in the described deterministic way as serial constraint elimination. The algorithm for path coverage is interpreted as path merging without any interpolation (no path constraint implies any other path constraint because they differ in at least one branch, therefore no merging in possible). Merge formulas are yielded from interpolation. With fewer constraints in merge formulas, more paths imply the merge formula and are pruned from the execution tree. The achieved coverage is given by the set of remaining paths.

Figure 3.11 illustrates four interesting algorithms and corresponding coverage. Unsat-cores are not unique. This corresponds to different path sets that can achieve the same coverage criterion. The left side of the figure considers the analyzed path sets and the corresponding code coverage. 'Contains' in arrow direction means that the set of analyzed program paths for one algorithm contains the set of analyzed program paths for the other one. The right side of the figure considers constraint sets. The set of constraints from the path constraint that remain after interpolation contains the set of constraints for another algorithm in the direction of the 'contains'-arrow. For the merge formulas (conjunction of interpolants) this means that a merge formula implies the merge formula of another algorithm in 'contains'-arrow direction, if both were computed in the same context for the same location.

Branch coverage (Backtracked) unsat-cores of unsatisfiable branches are used as merge formulas. A path is only pruned if it implies the previously computed backtracked unsat-cores. This means, that any

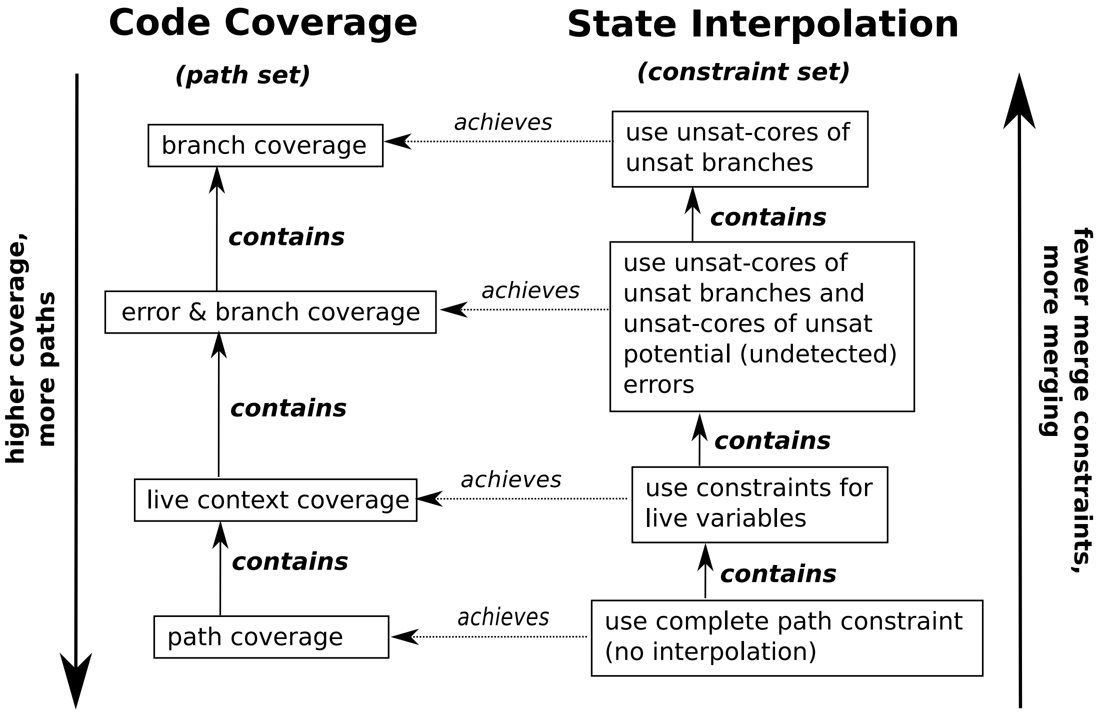

Code Coverage
(path set)
State Interpolation
(constraint set)
higher coverage, more paths
fewer merge constraints, more merging
branch coverage
achieves
use unsat-cores of unsat branches
contains
contains
error & branch coverage
achieves
use unsat-cores of unsat branches and unsat-cores of unsat potential (undetected) errors
contains
contains
live context coverage
achieves
use constraints for live variables
contains
contains
path coverage
achieves
use complete path constraint (no interpolation)

Figure 3.11 – Coverage and interpolation hierarchy [Ibi16a]

extension of the (pruned) path can not cover any yet uncovered branch. Therefore, this interpolation achieves branch coverage. Branch coverage means, that every branch in the program that can be covered with any program input is actually covered.

Error and branch coverage This is the algorithm described in Section 3.6.3. It uses unsat-cores for unsatisfiable branches and additionally unsat-cores of potential (and yet undetected) errors. This comprises unsat-cores necessary to achieve branch coverage. The additional constraints require to only prune a path when all previously unsatisfiable error conditions remain unsatisfiable. This means that any extension of the pruned path can not witness any yet undetected error. Error coverage means, that every error that is satisfiable with any program input, and for whose potential existence a constraint is generated, is actually witnessed on a remaining (not pruned) path.

Live context coverage In backtracking, the sets of dead and live variables are exactly known. Live variables are the ones that are read on at least one extension of the current path. Only live variables can contribute to unsat-cores in a path extension. This interpolation therefore comprises the interpolation needed to achieve error and branch coverage. By not generating constraints for dead variables, path constraints can become identical. Context coverage means that every program location is covered in every distinct live context.

Path coverage Complete path constraints (including constraints for dead variables) are used, no interpolation is done. Every path constraint is different, so no paths are pruned. Depth-first traversal without path merging achieves path coverage, i.e., every satisfiable program path is actually covered. This includes (live) context coverage.

3.6.7 Relation to Abstract Interpretation

Abstract interpretation was shortly reviewed in the introduction on page 5, an overlap with certain other program analysis approaches including symbolic execution was outlined on page 6. While symbolic execution traverses (parts of) the execution tree, abstract interpretation performs a fixed-point iteration on a graph in an abstract domain. The abstraction yields a finite state system (a finite graph) and is typically computed with predicate abstraction. A prominent tool example is PVS [GS97]. In order to speed up convergence of the fixed-point iteration, abstract interpretation uses Widening and Narrowing operations [CC77]. Abstract interpretation can be applied for bug detection without false negatives, but typically has false positives. A method to iteratively remove false positives is counterexample-guided abstraction refinement (CEGAR [CGL+03]). It consists in adding predicates to remove spurious counterexamples.

This chapter has illustrated that path merging folds the execution tree into a graph. This makes it interesting to consider the relationship to abstract interpretation again. A program path through a loop contains the same CFG node multiple times. In the backtracking phase of depth-first symbolic execution with path merging, merge formulas are assigned to such CFG nodes multiple times (one assignment each time when backtracking reaches the node). A later assignment assigns a more generalized (abstracted) formula. This can be seen as an instance of widening in abstract interpretation. Further, interpolation of unsatisfiable paths, interpolation at program end and interpolation of unsatisfiable potential error locations can be seen as widening in a similar way: abstractions (in the form of path constraints) are weakened by removing constraints.

Some differences remain. Symbolic execution does not explore unsatisfiable paths. Unsat-cores limit the widening of the abstraction. This can be used to limit widening to comprise only error-free paths (using unsat-cores for unsatisfiable paths and unsatisfiable error conditions, subsection 3.6.3), or to comprise only paths with limited coverage (unsat-cores for unsatisfiable paths, subsection 3.6.4). Further, the work at hand considers only finite execution trees and in contrast to abstract interpretation does not consider completeness.

Chapter 4

Automated Error Detection

Interesting bugs are context-sensitive, and a context-sensitive analysis is needed to accurately detect them. This chapter describes automated bug detection during symbolic execution, where the aim is to detect bugs as accurately as possible within one (partial) traversal of the execution tree. Detection of bugs that depend on symbolic input requires a solver, while detection of bugs that only depend on concrete input does not.

The work at hand combines static and dynamic checks in order to detect both bugs depending on symbolic input and bugs depending only on concrete input. The former type of bugs is detected during symbolic interpretation, while the latter type is detected with binary instrumentation during concrete execution.

Consecutive faults are not to be detected. When a bug is detected, it is reported. Then, the bug condition is negated and added to the path constraint, in order to find additional bugs on extensions of the current path. A special case are data races. If a data race occurs, then negating the bug condition means pruning the current path. Therefore, FIFO scheduling on one CPU core is used (Section 3.1). This prevents races, while many race conditions can still be detected (Section 4.3.5).

Detection of bugs that depend on symbolic input with the SMT solver during symbolic interpretation is described in Section 4.2, detection of bugs that only depend on concrete input during the concrete execution phases of selective symbolic execution with binary instrumentation is described in Section 4.3.

4.1 Test Programs and Modelled Part of Standard Library

4.1.1 Modelled Part of Standard Library

The usage of function models in general is described on pages 19ff. The following functions are modelled.

C standard library getenv, memcpy, rand, srand, printf, fprintf, fwprintf, fclose, fwgets, fgetws, fopen, fgets, fscanf, gets, puts, strcpy, strlen, wcscpy, wcslen, time, atoi, bind, listen, accept, connect

Posix pthread_create, pthread_exit, pthread_join, pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_unlock

Windows LogonUserA, LogonUserW, CryptAcquireContext, CryptCreateHash, CryptHashData, CryptDeriveKey, CryptEncrypt

Functions from the C standard library are either modelled to simply generate unconstrained input. This is the case for rand() and fgets(), for example. The function model just instantiates symbolic variable(s) without setting constraints on them. For other functions, a model needs to instantiate symbolic variable(s) with constraints that depend on symbolic function parameters, in order to enable accurate bug detection. This is the case for memcpy(), for example. The formula (constraint) of the destination buffer depends on the symbolic source buffer.

The models for pthread functions like pthread_create() and pthread_exit() manage the symbolic stack objects for thread operations. For thread creation, a new symbolic stack object is instantiated to enable symbolic interpretation. The mutex functions were modelled for a first experiment on data race detection during symbolic interpretation, before this was removed and ThreadSanitizer [SI09] was used for more efficient data race detection during concrete execution [Ibi16b]. Data race detection is discussed on page 84ff.

Windows cryptographic functions and standard library functions that perform input/output operations were modelled for a first experiment on information exposure detection during symbolic interpretation. The models set information flow labels that identify sensitive information like password or keys (models of cryptographic functions) or environment variables (model of getenv() from the standard library) in order to enable detection of information exposures by propagating these labels through the data flow. These labels are checked in the function models of output functions like printf(). Information exposure detection is discussed on page 83f.

4.1.2 Test Programs from Juliet Suite

The following tests from the Juliet Suite [Uni13] are used:

- CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy
- CWE121_Stack_Based_Buffer_Overflow__CWE129_fgets
- CWE366_Race_Condition_Within_Thread__global_int
- CWE366_Race_Condition_Within_Thread__int_byref
- CWE835_Infinite_Loop__do
- CWE835_Infinite_Loop__do_true
- CWE835_Infinite_Loop__for
- CWE835_Infinite_Loop__for_empty
- CWE835_Infinite_Loop__while
- CWE835_Infinite_Loop__while_true
- CWE190_Integer_Overflow__char_fscanf_add
- CWE190_Integer_Overflow__char_fscanf_multiply
- CWE190_Integer_Overflow__char_fscanf_square
- CWE190_Integer_Overflow__char_max_add
- CWE190_Integer_Overflow__char_max_multiply
- CWE190_Integer_Overflow__char_max_square
- CWE190_Integer_Overflow__char_rand_add
- CWE190_Integer_Overflow__char_rand_multiply
- CWE190_Integer_Overflow__char_rand_square
- CWE190_Integer_Overflow__int64_t_fscanf_add
- CWE190_Integer_Overflow__int64_t_fscanf_multiply
- CWE190_Integer_Overflow__int64_t_fscanf_square
- CWE190_Integer_Overflow__int64_t_max_add
- CWE190_Integer_Overflow__int64_t_max_multiply
- CWE190_Integer_Overflow__int64_t_max_square
- CWE190_Integer_Overflow__int64_t_rand_add
- CWE190_Integer_Overflow__int64_t_rand_multiply
- CWE190_Integer_Overflow__int64_t_rand_square
- CWE190_Integer_Overflow__int_connect_socket_add
- CWE190_Integer_Overflow__int_connect_socket_multiply
- CWE190_Integer_Overflow__int_connect_socket_square
- CWE190_Integer_Overflow__int_fgets_add
- CWE190_Integer_Overflow__int_fgets_multiply
- CWE190_Integer_Overflow__int_fgets_square
- CWE190_Integer_Overflow__int_fscanf_add

— CWE190_Integer_Overflow__int_fscanf_multiply

— CWE190_Integer_Overflow__int_fscanf_square

— CWE190_Integer_Overflow__int_listen_socket_add

— CWE190_Integer_Overflow__int_listen_socket_multiply

— CWE190_Integer_Overflow__int_listen_socket_square

— CWE190_Integer_Overflow__int_max_add

— CWE190_Integer_Overflow__int_max_multiply

— CWE190_Integer_Overflow__int_max_square

— CWE190_Integer_Overflow__int_rand_add

— CWE190_Integer_Overflow__int_rand_multiply

— CWE190_Integer_Overflow__int_rand_square

— CWE190_Integer_Overflow__short_fscanf_add

— CWE190_Integer_Overflow__short_fscanf_multiply

— CWE190_Integer_Overflow__short_fscanf_square

— CWE190_Integer_Overflow__short_max_add

— CWE190_Integer_Overflow__short_max_multiply

— CWE190_Integer_Overflow__short_max_square

— CWE190_Integer_Overflow__short_rand_add

— CWE190_Integer_Overflow__short_rand_multiply

— CWE190_Integer_Overflow__short_rand_square

— CWE190_Integer_Overflow__unsigned_int_fscanf_add

— CWE190_Integer_Overflow__unsigned_int_fscanf_multiply

— CWE190_Integer_Overflow__unsigned_int_fscanf_square

— CWE190_Integer_Overflow__unsigned_int_max_add

— CWE190_Integer_Overflow__unsigned_int_max_multiply

— CWE190_Integer_Overflow__unsigned_int_max_square

— CWE190_Integer_Overflow__unsigned_int_rand_add

— CWE190_Integer_Overflow__unsigned_int_rand_multiply

— CWE190_Integer_Overflow__unsigned_int_rand_square

— CWE259_Hard_Coded_Password__w32_char

— CWE325_Missing_Required_Cryptographic_Step__w32_CryptCreateHash

— CWE325_Missing_Required_Cryptographic_Step__w32_CryptDeriveKey

— CWE325_Missing_Required_Cryptographic_Step__w32_CryptEncrypt

— CWE325_Missing_Required_Cryptographic_Step__w32_CryptHashData

— CWE666_Operation_on_Resource_in_Wrong_Phase_of_Lifetime__accept_bind_listen

— CWE666_Operation_on_Resource_in_Wrong_Phase_of_Lifetime__accept_listen_bind

— CWE666_Operation_on_Resource_in_Wrong_Phase_of_Lifetime__bind_accept_listen

— CWE666_Operation_on_Resource_in_Wrong_Phase_of_Lifetime__listen_accept_bind

— CWE666_Operation_on_Resource_in_Wrong_Phase_of_Lifetime__listen_bind_accept

— CWE526_Info_Exposure_Environment_Variables__basic

— CWE534_Info_Exposure_Debug_Log__w32_char

— CWE534_Info_Exposure_Debug_Log__w32_wchar_t

— CWE535_Info_Exposure_Shell_Error__w32_char

— CWE535_Info_Exposure_Shell_Error__w32_wchar_t

The Juliet Suite combines baseline bugs with different control flow and data flow variants [Uni13]. The flow variants for C are listed in Table 4.1.

4.2 Error Detection During Symbolic Interpretation

Potential error locations that may directly depend on symbolic input are determined during static pre-analysis (Section 2.4), and breakpoints are set on these locations. When the debugger breaks at such a location, a bug condition is generated. When the path constraint together with the bug condition are satisfiable, then the bug with corresponding input is reported. The solver call with path constraint P and error condition E is:

$$\text{check-sat}(\ P \wedge E\ )$$

If the solver returns *true*, then the detected error is reported.

4.2.1 Memory Access Errors

Potential error locations are the following expression types with at least one symbolic parameter:

— array subscript expressions

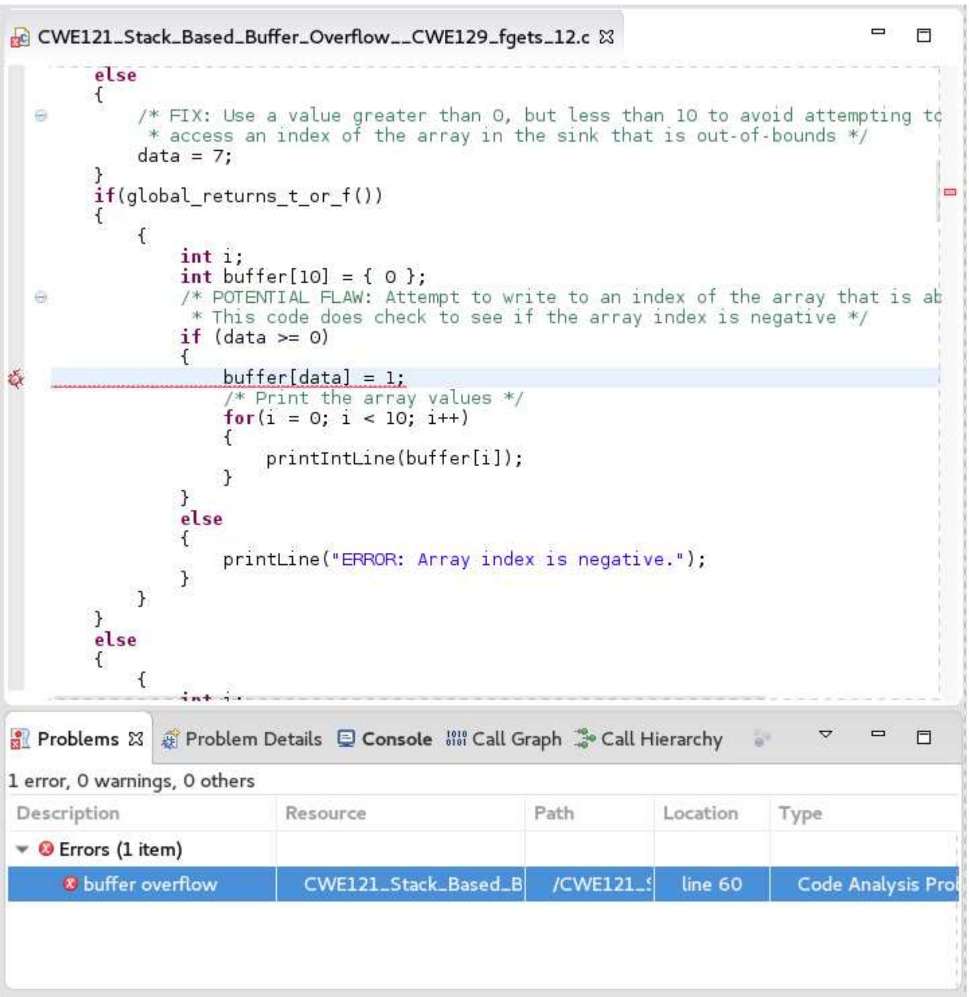
— pointer dereference

Checks are triggered during interpretation of such expressions. An array subscript expression contains an array expression (the array) and a subscriptexpression (the index). The size of the array (buffer) is denoted as b_S, the integer index is denoted as i. Both may have concrete or symbolic values, i.e., they may be logic formulas. A pointer is modelled by the symbolic interpreter as having a target and an integer offset. The size of the pointer target is also denoted as b_S, the integer offset is also denoted as i. A buffer underread or buffer underwrite (CWE-127,124) means that $i < 0$ is satisfiable, i.e., the index/offset can be negative. The error condition is:

$$E_{\text{bufferunderflow}} = (i < 0)$$

Flow variant	Description
1	Baseline - Simplest form of the flaw
2	if(1) and if(0)
3	if(5==5) and if(5!=5)
4	if(STATIC_CONST_TRUE) and if(STATIC_CONST_FALSE)
5	if(staticTrue) and if(staticFalse)
6	if(STATIC_CONST_FIVE==5) and if(STATIC_CONST_FIVE!=5)
7	if(staticFive==5) and if(staticFive!=5)
8	if(staticReturnsTrue()) and if(staticReturnsFalse())
9	if(GLOBAL_CONST_TRUE) and if(GLOBAL_CONST_FALSE)
10	if(globalTrue) and if(globalFalse)
11	if(globalReturnsTrue()) and if(globalReturnsFalse())
12	if(globalReturnsTrueOrFalse())
13	if(GLOBAL_CONST_FIVE==5) and if(GLOBAL_CONST_FIVE!=5)
14	if(globalFive==5) and if(globalFive!=5)
15	switch(6) and switch(7)
16	while(1)
17	for loops
18	goto statements
21	Flow controlled by value of a static global variable. All functions contained in one file.
22	Flow controlled by value of a global variable. Sink functions are in a separate file from sources.
31	Data flow using a copy of data within the same function
32	Data flow using two pointers to the same value within the same function
34	Use of a union containing two methods of accessing the same data (within the same function)
41	Data passed as an argument from one function to another in the same source file
42	Data returned from one function to another in the same source file
44	Data passed as an argument from one function to a function in the same source file called via a function pointer
45	Data passed as a static global variable from one function to another in the same source file
51	Data passed as an argument from one function to another in different source files
52	Data passed as an argument from one function to another to another in three different source files
53	Data passed as an argument from one function through two others to a fourth; all four functions are in different source files
54	Data passed as an argument from one function through three others to a fifth; all five functions are in different source files
61	Data returned from one function to another in different source files
63	Pointer to data passed from one function to another in different source files
64	void pointer to data passed from one function to another in different source files
65	Data passed as an argument from one function to a function in a different source file called via a function pointer
66	Data passed in an array from one function to another in different source files
67	Data passed in a struct from one function to another in different source files
68	Data passed as a global variable in the ”a” class from one function to another in different source files

Table 4.1 – The Juliet suite combines baseline bugs with data and control flow variants [Uni13]



```
    else
    {
        /* FIX: Use a value greater than 0, but less than 10 to avoid attempting tc
         * access an index of the array in the sink that is out-of-bounds */
        data = 7;
    }
    if(global_returns_t_or_f())
    {
        {
            int i;
            int buffer[10] = { 0 };
            /* POTENTIAL FLAW: Attempt to write to an index of the array that is ab
             * This code does check to see if the array index is negative */
            if (data >= 0)
            {
                buffer[data] = 1;
                /* Print the array values */
                for(i = 0; i < 10; i++)
                {
                    printIntLine(buffer[i]);
                }
            }
            else
            {
                printLine("ERROR: Array index is negative.");
            }
        }
    }
    else
    {
        {
            int i;
```

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
buffer overflow	CWE121_Stack_Based_B	/CWE121_S	line 60	Code Analysis Pro

Figure 4.1 – Screenshot error reporting

A buffer overread (CWE-126) or buffer buffer overwrite (overflow; CWE-121,122) means that $i \geq b_S$ is satisfiable, i.e., the index/offset can be equal to or larger than the buffersize. The error condition is:

$$E_{\text{bufferoverflow}} = (i >= b_S)$$

Examples for buffer overflows from the Juliet suite and their detection are discussed on page 50 in Section 3.6.3. A screenshot of error reporting in the CDT GUI is shown in Figure 4.1.

4.2.2 Number Format Errors

Potential error locations are the following expression types with at least one symbolic parameter:

— arithmetic operations

— type casts

Arithmetic expressions are unary expressions (operator and one expression) or binary expressions (operator and two expressions). A prominent bug type are integer overflows or wrap arounds (CWE-190). The error condition is:

$$E_{\text{numoverflow}} = (expr > MAX)$$

where *expr* is the translated arithmetic expression and MAX is the maximum value of the result type (concrete value). Correspondingly, the error condition for integer underflows (CWE-191) is:

$$E_{\text{numunderflow}} = (expr < MIN)$$

where MIN is the minimum value of the result type (concrete value). Signed to unsigned conversion errors or unsigned to signed conversion errors (CWE-195,196) are detected along the same line. The error conditions for number typecasts are:

$$E_{\text{castoverflow}} = (expr > MAX)$$

$$E_{\text{castunderflow}} = (expr < MIN)$$

where MAX is the maximum value of the result type and MIN is the minimum value of the result type. A related bug type is division by zero (CWE-369). Potential error locations are division expressions (binary expression with division operator) with symbolic divisor expression. For detection, the solver is queried

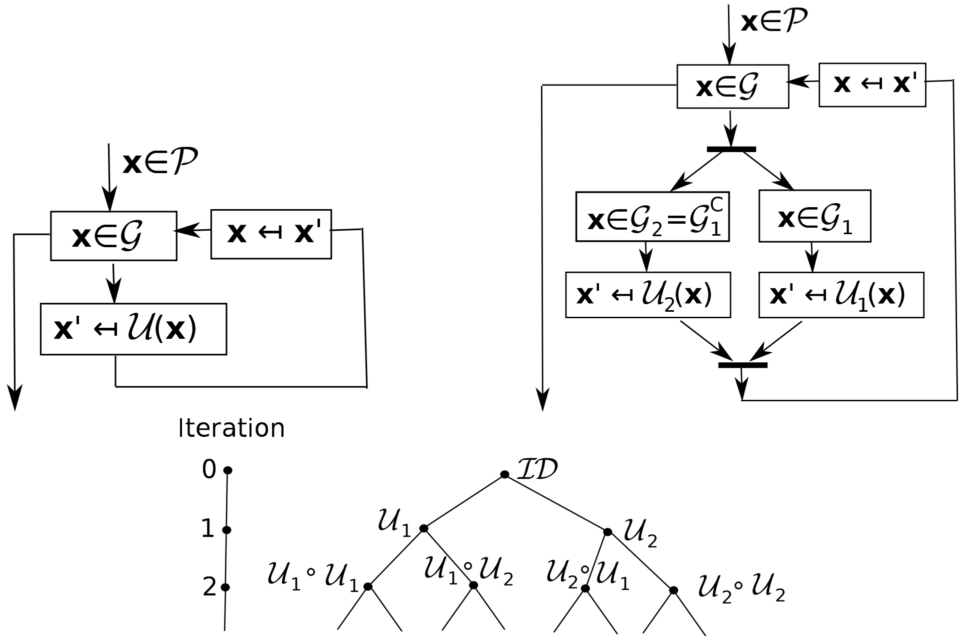

$x \in \mathcal{P}$
$x \in \mathcal{G}$
$x \hookleftarrow x'$
$x' \hookleftarrow \mathcal{U}(x)$
$x \in \mathcal{P}$
$x \in \mathcal{G}$
$x \hookleftarrow x'$
$x \in \mathcal{G}_2 = \mathcal{G}_1^C$
$x \in \mathcal{G}_1$
$x' \hookleftarrow \mathcal{U}_2(x)$
$x' \hookleftarrow \mathcal{U}_1(x)$
Iteration
0
1
2
$\mathcal{ID}$
$\mathcal{U}_1$
$\mathcal{U}_2$
$\mathcal{U}_1 \circ \mathcal{U}_1$
$\mathcal{U}_1 \circ \mathcal{U}_2$
$\mathcal{U}_2 \circ \mathcal{U}_1$
$\mathcal{U}_2 \circ \mathcal{U}_2$

Figure 4.2 – Single-path loop, multi-path loop and transformation monoid [IM15]

whether a divisor of zero is satisfiable under the path constraint. The error condition is:

$$E_{\text{divzero}} = (expr = 0)$$

where *expr* is the divisor expression.

4.2.3 Infinite Loops

Some infinite loops are intended by the developer, for example in reactive systems. On the other hand, input-dependent infinite loops are often unintended, i.e., bugs. Infinite loop bugs are both an issue of software safety and of software security. A program that runs into an infinite loop bug becomes unresponsive, which violates safety properties. If an infinite loop can be triggered with program input by an attacker, the program is vulnerable to a denial of service attack. Under the common weakness enumeration, infinite loops are known as CWE-835 ('loop with unreachable exit condition') or 'unchecked input for loop condition' (CWE-606).

Potential error locations are (revisited) decision nodes that depend on symbolic input. Loops can be categorized into single-path loops, multi-path and nested loops. A loop model is illustrated for a single-path loop in figure 4.2 (upper left). The figure shows the vector $\mathbf{x}$ of loop variables, a path constraint $\mathcal{P}$ on which the loop is reached from the program entry point, the loop guard set $\mathcal{G}$ and the loop update function $\mathcal{U}(\mathbf{x})$. The vector $\mathbf{x}'$ denotes the loop variables after update, so that one loop update computes

$$\mathbf{x}' = \mathcal{U}(\mathbf{x})$$

The problem of interest is to detect all loops, that may run infinitely for any program input. This includes the question whether such a loop can be reached on a satisfiable path with corresponding input.

A loop may run a different number of iterations for different input. This means that it computes different functions for different sets of input. The sequence $(\mathbf{x}^{(k)})_{k\in\mathbb{N}}$ of values of $\mathbf{x}$ for each iteration is denoted as the orbit of $\mathbf{x}$. The loop effect is described with function composition of the update functions of different iterations. If there is input for the single-path loop from figure 4.2 for which it runs n iterations, then it computes the orbit:

$$\begin{aligned}
\mathbf{x} = \mathbf{x}^{(0)} \quad &\mapsto \quad \mathbf{x}^{(1)} = \mathcal{U}(\mathbf{x}^{(0)}) \\
&\mapsto \quad \mathbf{x}^{(2)} = \mathcal{U}(\mathbf{x}^{(1)}) = \mathcal{U}^2(\mathbf{x}^{(0)}) \\
&\mapsto \quad \dots \\
&\mapsto \quad \mathbf{x}^{(n)} = \mathcal{U}^n(\mathbf{x})
\end{aligned}$$

A multi-path loop contains different loop update functions for each path. A two-path loop is illustrated in Figure 4.2 (upper right). Different paths through the loop in each iteration may be taken for different input sets. The possible compositions of update functions form a transformation monoid, which is also illustrated in Figure 4.2 (bottom). If the loop is not taken (0 iterations), this corresponds to the identity mapping $\mathcal{ID}$ of $\mathbf{x}$. For one iteration the resulting function is either $\mathcal{U}_1$ or $\mathcal{U}_2$. For a two-path loop (e.g. an `If/Else` statement within a loop) the transformation monoid is a binary tree. The problem is to detect whether a loop can be reached with adequate input so that an infinite orbit is satisfied.

This subsection describes an algorithm, that is based on fixed-point satisfiability checks with the solver and that makes use of Brouwer's fixed-point theorem [Bro11] to speed-up analysis in case of 'simple' loops. Brouwer's theorem states that any continuous mapping of a compact (bounded and closed) convex set into itself has a fixed-point. The number formats are discrete and finite, so that an infinite orbit (without overflow) must be a periodic orbit, i.e.

$$\mathbf{x}^{(t+p)} = \mathbf{x}^{(t)} \quad \text{for some } p > 0,\ t \geq 0$$

The symbolic execution engine keeps the program path in memory which is symbolically interpreted at the moment. A loop is reached with symbolic execution from a program entry point on a path which we call stem path (as in [GRH$^+$08]). When an already visited decision node (which is on the current path) is visited again, the algorithm to test for an infinite loop is called. The path is split into stem and loop body at each previous occurrence of the loop decision node. Loop variables x_i' from the loop body and the corresponding x_i from the stem as their previous values are identified.

The algorithm performs two types of tests. The first type is a context-free analysis to decide whether 'simple' loops terminate for all input, or whether they are infinite for all input. The context-free analysis is performed only once per loop statement. The second type performs context-sensitive non-termination analysis of the loops which have not yet been decided in the first step. Both types are run during symbolic execution. The method for context-free checks is given as pseudo-code in Algorithm 4. The method for context-sensitive non-termination checks is depicted as pseudo-code in Algorithm 5.

Context-free (Non-)Termination Tests for 'Simple' Loops

Loop property determination The three properties of interest are whether the loop is single-path, whether it is linear with convex guard set (only linear arithmetic), and whether it contains the modulo operator. There is an AST subtree corresponding to the loop compound statement, which is accessible through a reference from the CFG decision node. This AST subtree is traversed once with an AST visitor. This can be done during static pre-analysis (Section 2.4), or when the decision node is revisited for the first time on any path. The property determination needs to run only once per loop statement.

The context-free tests are performed only once per loop statement, when the loop decision node is revisited for the first time on any path. Stem path equations are not included in the equation system, which removes the context of the loop. The remaining constraints are the guard condition before the loop, the loop update and the guard condition after the update. Any recomputations later when the loop is reached with a different path constraint would therefore be redundant. The algorithm generates an additional constraint for termination and non-termination checks respectively and tests equation system satisfiable with the SMT solver (compare Algorithm 4).

Termination This check is performed for single-path loops with linear arithmetic and convex guard set, which do not contain the modulo operator (which would destroy the precondition of Brouwer's theorem).

The loop guard can be expressed in matrix notation as:

$$G\mathbf{x} \leq \mathbf{g}, \qquad G \in \mathbb{R}^{k\times n},\ \mathbf{g} \in \mathbb{R}^n$$

The loop guard is a convex set if it consists of conjunctively connected linear inequalities (no disjunctions). The loop update function is then:

$$\mathbf{x}' = \mathcal{U}(\mathbf{x}) = U\mathbf{x} + \mathbf{u}, \qquad U \in \mathbb{R}^{n\times n},\ \mathbf{x}, \mathbf{x}', \mathbf{u} \in \mathbb{R}^n$$

The loop always terminates if the orbit leaves the guard set for all possible input vectors $\{\mathbf{x} \mid G\mathbf{x} \leq \mathbf{g}\}$ after a finite number of iterations. In the paragraph on soundness and bounded completeness on page 68 it is explained in detail that as a consequence of Brouwer's fixed-point theorem [Bro11] there can only be a periodic orbit if the loop update function $\mathcal{U}(\mathbf{x})$ contains a fixed-point $\mathbf{x}^{(1)} = \mathbf{x}^{(0)}$ within the guard set:

$$\begin{aligned} & G\mathbf{x} \leq \mathbf{g} \\ \wedge\quad & \mathbf{x}' = U\mathbf{x} + \mathbf{u} \\ \wedge\quad & \mathbf{x}' = \mathbf{x} \end{aligned}$$

If the constraint solver answers `'unsat'` to this query, the loop is marked as terminating.

Non-termination This check is performed for single-path loops (possibly non-linear). If the loop was already decided to always terminate, then this check is not performed. The loop is non-terminating for all input if:

$$\begin{aligned} & \forall(\mathbf{x} \in \mathcal{G}).\ \mathcal{U}(\mathbf{x}) \in \mathcal{G} \\ \Leftrightarrow\quad & \neg\exists(\mathbf{x}_i \in \mathcal{G}).\ \mathcal{U}(\mathbf{x}_i) \notin \mathcal{G} \end{aligned}$$

The constraint solver is therefore asked to check satisfiability of:

$$\mathbf{x} \in \mathcal{G}\ \wedge\ \mathbf{x}' = \mathcal{U}(\mathbf{x})\ \wedge\ \neg(\mathbf{x}' \in \mathcal{G}) \tag{4.1}$$

If the solver answers `'unsat'` to this query, the loop is marked as non-terminating.

```
checkContextFree(CFNode node, SymVars $\mathbf{x}$, SymVars $\mathbf{x}'$)
if (isLinearNoModulo(node)) then
    // 1. check terminating:
    s1 = check-sat$\big(G\mathbf{x} \leq \mathbf{g} \wedge \mathbf{x}' = U\mathbf{x} + \mathbf{u} \wedge \mathbf{x}' = \mathbf{x}\big)$;
    if (s1 == unsat) then
        markTerminating(node);
        return;
if (isSinglePath(node)) then
    // 2. check infinite:
    s2 = check-sat$\big(\mathbf{x} \in \mathcal{G} \;\wedge\; \mathbf{x}' = \mathcal{U}(\mathbf{x}) \;\wedge\; \neg(\mathbf{x}' \in \mathcal{G})\big)$;
    if (s2 == unsat) then
        markNonTerminating(node);
        return;
```

Algorithm 4: Context-free termination and non-termination checks for 'simple' loops [IM15].

Context-sensitive Non-Termination Tests for Remaining Loops

Symbolic execution automatically explores the satisfiable part of the transformation monoid, given the current path constraint. Loops which have not been decided by the context-free analysis are analyzed context-sensitively during symbolic execution. Such loops are e.g. multi-path loops or loops which terminate only for a partial input set. If the loop guard is satisfiable for the stem path context, then symbolic execution unrolls the loop once and checks guard satisfiability again. It is detected that a loop is closed when the same decision node on the path is revisited (the unrolled loop body is then denoted as 'closed loop'). If the guard condition is satisfiable also after unrolling the loop body, then the loop variables are identified and the non-termination analysis algorithm is called. If the same decision node is already several times on the path, then several loops are closed at the same time. The test is then run for each of the closed loops. The test checks the existence of a fixed-point in the unrolled loop. This corresponds to checking for a satisfiable periodic orbit in the respective composite function of the transformation monoid, that corresponds to the taken loop unrolling branches. A fixed-point of, e.g., $\mathcal{U}_1^4$ is a periodic orbit of $\mathcal{U}_1$ with periodicity 4. A fixed-point of an n-fold function composition from the transformation monoid is a periodic orbit through the loop with periodicity $p = n$. The loop body path constraint contains the constraints for correct domains of the respective update functions. By checking for fixed-points for all loop closed events on the current path, also periodic orbits with t prefix iterations and periodicity $p = n - t$ are found. The fixed-point equations for the loop variables are in SMTLIB

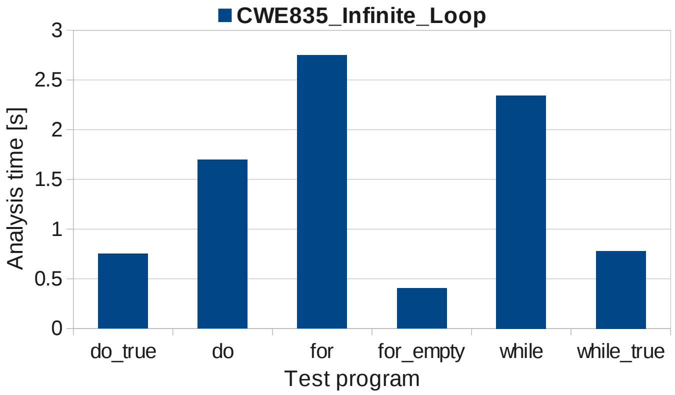


Figure 4.3 – Analysis run-times for infinite loop test programs [IM15]

notation:

$$\texttt{(assert (=}\ x_1\texttt{'}\ x_1\ \texttt{))}$$

$$\ldots$$

$$\texttt{(assert (=}\ x_n\texttt{'}\ x_n\ \texttt{))}$$

where the x_i' and x_i are replaced with the actual (translated) loop variable names in single assignment form. If the solver answers SAT, an infinite loop is detected and reported. Corresponding input triggering the infinite loop bug can be queried from the solver with the SMTLIB `'get-value'` command.

```
checkContextSens(CFNode node, SymVars $\mathbf{x}$, SymVars $\mathbf{x}'$)
$\mathcal{P}(\mathbf{x})$ = getPathConstraint();
$\mathcal{L}(\mathbf{x}')$ = getUnrolledLoopConstraint();
s = check-sat$\big(\mathcal{P}(\mathbf{x}) \wedge \mathcal{L}(\mathbf{x}') \wedge (\mathbf{x}' == \mathbf{x})\big)$;
if (s == sat) then
    reportError(node, 'infinite Loop');
    return;
else
    return;
```

Algorithm 5: Context-sensitive non-termination check for remaining loops [IM15].

Soundness

Proposition 1. *The algorithm does not compute any false positive infinite loop detections.*

Proof. The detection algorithm is constructive. An infinite loop is reported in two cases:

Case 1. *by the context-free check if:*

$$\textit{check-sat}\Big(\mathcal{G}(\mathbf{x}) \land (\mathbf{x}' = \mathcal{U}(\mathbf{x})) \land \neg(\mathcal{G}(\mathbf{x}'))\Big) == \texttt{unsat}$$

i.e. there exists no loop input $\mathbf{x}_i \in \mathcal{G}(\mathbf{x})$ for which the next iteration is not also taken. This means that the loop is infinite if it can be reached with any input. The check function is only called if the loop is reached with satisfiable input during symbolic execution. The program therefore indeed contains a reachable infinite loop.

Case 2. *by the context-sensitive check if:*

$$\textit{check-sat}\Big(\mathcal{P}(\mathbf{x}) \land \mathcal{L}(\mathbf{x}') \land (\mathbf{x}' == \mathbf{x})\Big) == \texttt{sat}$$

i.e. there is a program input for which the loop is reached with $\mathcal{P}(\mathbf{x}_i)$, so that loop guard and an unrolled loop body are satisfiable and have a fixed-point $\mathcal{L}(\mathbf{x}_i)$. The program therefore indeed contains a reachable infinite loop.

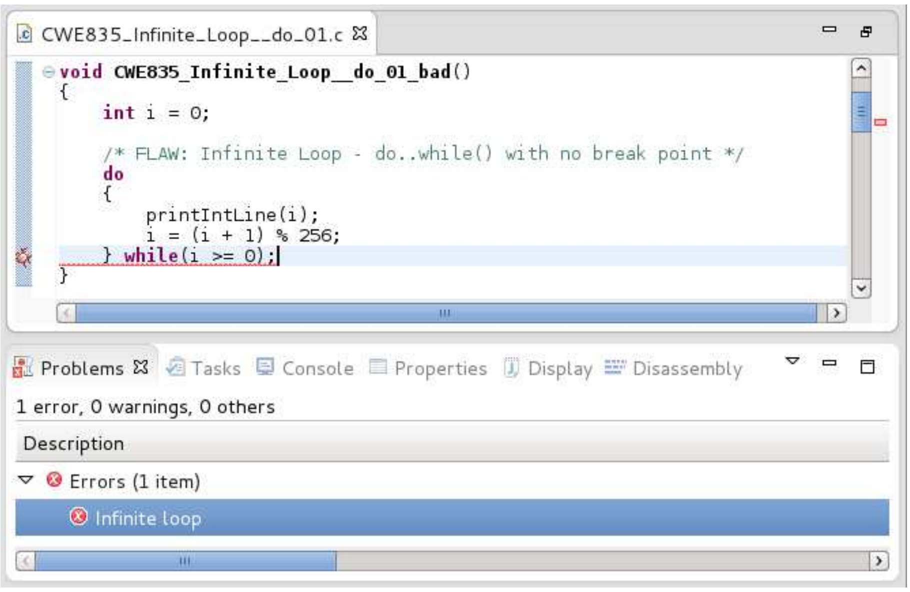
□

Bounded completeness

In order to prove bounded completeness, it must first be ensured that the context-free termination check never wrongly marks a loop as terminating if the loop does not always terminate. Otherwise this would prevent the context-sensitive non-termination check from being performed and would lead to false negative detections. This means:

Proposition 2. *The context-free termination check is sound, i.e., when a context-free single-path loop with convex guard set and linear update function without modulo operator does not have a fixed-point, then there is no infinite orbit through it with any stem path constraint.*

Proof. By contradiction. The proof is a minor adaptation from [HCS06]. Assume there is a periodic orbit $(\mathbf{x}^{(k)})$, $k \in \mathbb{N}_0$, through the loop, i.e. $\forall k.\ \mathbf{x}^{(k)} \in \mathcal{G}.\ \land\ \mathbf{x}^{(t+p)} = \mathbf{x}^{(t)}$. Without loss of generality, $t = 0$ can be assumed, i.e., k loop iterations belong to the unrolled loop body and there are no prefix iterations. Let $\mathcal{V}$ be the affine space of the lowest dimension that contains the orbit. By intersecting $\mathcal{G}$ and $\mathcal{V}$, a compact convex set $\mathcal{G} \cap \mathcal{V}$ is yielded. Furthermore $\mathbf{x}^{(k)} \in \mathcal{G} \cap \mathcal{V};\ 0 \leq k \leq p$. The convex hull $\mathcal{X} = \mathrm{Conv}(x^{(k)})$ in considered. It is $\mathcal{X} \subset \mathcal{V} \cap \mathcal{G}$ and $\forall \mathbf{x}_i \in \mathcal{X}, \forall q \in \mathbb{N} : \mathcal{U}^q(\mathbf{x}_i) \in \mathcal{X}$, i.e. $\mathcal{X}$ is a 'positively invariant set'. The closure $\overline{\mathcal{X}}$ is a compact convex positively invariant set in $\mathcal{V}$. Now the Brouwer fixed-point



```
void CWE835_Infinite_Loop__do_01_bad()
{
    int i = 0;

    /* FLAW: Infinite Loop - do..while() with no break point */
    do
    {
        printIntLine(i);
        i = (i + 1) % 256;
    } while(i >= 0);
}
```

1 error, 0 warnings, 0 others

Description

Errors (1 item)

Infinite loop

Figure 4.4 – Screenshot infinite loop detection [IM15]

theorem [Bro11] can be applied on the loop update function $\mathcal{U}\colon \overline{X} \to \overline{X}$. It guarantees the existence of a fixed-point in $\mathcal{U}$. □

Now bounded completeness can be considered:

Proposition 3. *When all loops are unrolled up to a depth of n (in all satisfiable contexts), then no infinite loop bug with t prefix iterations and an orbit periodicity of $p \leq n - t$ is missed.*

Proof. Unrolling all loops up to a depth of n (if satisfiable) means building all composition functions of the loop's transformation monoid up to level n. Any orbit with periodicity $p \leq n - t$ is a fixed-point of a composite function from the transformation monoid with level $l \leq n - t$ and is therefore not missed. □

Trading-off complexity versus accuracy

It is possible to reduce the computational effort for infinite loop detection at the expense of false negative detections. One can test for not all possible infinite loops, but for only a certain subset. For example, one can test only loops with restricted loop path length, by allowing a maximum of n branches backwards. Again, there is a trade-off between computational effort and detection accuracy.

Experiments

The implementation is validated with the infinite loop test programs from the Juliet test suite [BB12, Uni13]. These test programs correspond to CWE-835. Juliet's Test cases for the related CWE-606

```
1   static void good1(){
2       int i = 0;
3       while(i >= 0) {
4           // FIX: Add a break point if i=10
5           if (i == 10) {
6               break;
7           }
8           printIntLine(i);
9           i = (i + 1) % 256;
10      }
11  }
12
13  void CWE835_Infinite_Loop_for_empty_01_bad(){
14      int i = 0;
15      // FLAW: Infinite Loop
16      for(;;) {
17          printIntLine(i);
18          i++;
19      }
20  }
21
22  void CWE835_Infinite_Loop__for_01_bad() {
23      int i = 0;
24      // FLAW: Infinite Loop
25      for (i = 0; i >= 0; i = (i + 1) % 256) {
26          printIntLine(i);
27      }
28  }
```

Figure 4.5 – Example tests for detection of infinite loops from [Uni13]

('unchecked input for loop condition') are not used, because those loops are in fact all terminating (but possibly with a very large number of iterations). The Juliet suite (current version 1.2) contains six test programs for infinite loop detection. All contained infinite loops are input-independent. In order to test the algorithm, the engine is run statically, i.e., without debugger. All CFG nodes are interpreted. A different algorithm, that detects infinite loops during selective symbolic execution (via binary instrumentation) is discussed in section 4.3.3.

Example 'good' (terminating) and 'bad' functions (with infinite loop) are shown in Figure 4.5. The 'good' multi-path loop (line 1-11) is unrolled ten times during analysis, at which point termination is proven. The figure also contains a single-path empty for-loop (line 13-20) which is non-terminating for all input. The loop guard is simply 'true', and the bug is detected during the context-free non-termination check. The figure further contains another single-path for-loop which is non-terminating for all input and is decided with the context-free test. The context-free termination check is not performed in this case because the loop contains a modulo operator. Detection runtimes are shown in Figure 4.3. All 6 test programs are correctly decided within less than three seconds each. A screenshot of error reporting is shown in figure 4.4.

4.3 Error Detection During Concrete Execution

Instrumentation and dynamic analysis are used to detect bugs during concrete execution. Rather than performing checks during interpretation, the program under test is instrumented to make it check itself. Binary instrumentation adds instructions to collect relevant runtime information in shadow memory and to check potential bug conditions. This section describes the combination of symbolic execution and binary instrumentation. The SMT solver is not needed to decide the concrete bug conditions, but it is used to drive execution into interesting program paths. For most bug types, available binary instrumentation can be reused. Examples are the address sanitizer [SBPV12] or the thread sanitizer [SI09]. Both of them are integrated in compilers like `gcc`, so that the program under test can be instrumented during compilation. The work at hand uses the following available tools: address sanitizer, thread sanitizer and gcov. These tools do not know anything of symbolic or concolic execution. The sanitizers are used to detect bugs during concrete execution (between debugger breakpoints) with binary instrumentation. Address sanitizer detects memory access errors, Subsection 4.3.1. Thread sanitizer detects race conditions (Subsection 4.3.5), deadlocks and multiple (un)locks (Subsection 4.3.6). Gcov instrumentation is used to trace coverage of concrete execution and the combination with symbolic execution enables detection of dead code (Subsection 4.3.7). For the detection of infinite loops which are independent of symbolic input, a detection algorithm is developed in Subsection 4.3.3). The source code under test is compiled and statically linked with the sanitizer libraries. Breakpoints are set on the sanitizer error report functions. In case the debugger breaks at such a location, the bug is localized by following the call stack back into the source files of interest.

4.3.1 Memory Access Errors

The address sanitizer [SBPV12] instruments memory accesses and detects memory corruption bugs like buffer overflows (CWE-121, 122,124,126,127), write-what-where conditions (CWE-123) or accesses to dangling pointers (use-after-free, CWE-416) in many situations. In [SBPV12], it is reported that the instrumentation slows down execution by about 73% and increases memory usage about 3.4 times.

Experiments Detection accuracy is evaluated with 19 test programs from Juliet [Uni13], that contain buffer overflows on the stack with `memcpy()`. Figure 4.6 shows an example. The 'bad' function contains a 'baseline' bug (simplest flow variant) with `memcpy` in line 6. The bug is that the size of the complete struct is used where only the size of a contained array is meant. The debugger breaks on the address

```
void CWE121_memcpy_01_bad() {
  charvoid cv_struct;
  cv_struct.y = (void *)SRC_STR;
  // FLAW: overwrite the pointer y
  memcpy(cv_struct.x, SRC_STR,
    sizeof(cv_struct));
  cv_struct.x[(sizeof(cv_struct.x)/
    sizeof(char))-1]='\0';
  printLine((char *)cv_struct.x);
  printLine((char *)cv_struct.y);
}
```

Figure 4.6 – Example buffer overflow from [Uni13]

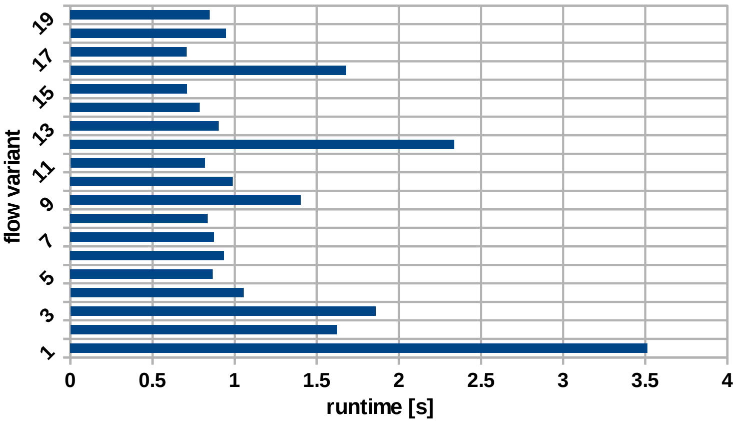


Figure 4.7 – Detection run-times for buffer overflow tests [Ibi15b]

sanitizer's error reporting function and the bug is correctly localized. For flow variant 9 ('control flow depending on global variables'), the address sanitizer first misses the overflow, but then detects it through reception of a segmentation fault signal from the operating system. The results are illustrated in Figure 4.7. All bugs are detected in about 1-2s each.

4.3.2 Number Format Errors

Current compilers like gcc also support instrumentation for detection of integer overflows and related errors ('undefined behaviour sanitizer'). The compiler inserts checks that are executed during concrete execution.

4.3.3 Infinite Loops

The detection of infinite loops that directly depend on symbolic input is discussed in Section 4.2.3 (in this case, the SMT solver is used). SMT solver checks for such infinite loops are only performed at revisited interpreted decision nodes.

The presence of an infinite loop that only depends on concrete input causes one of the two following effects during selective symbolic execution. Either the concrete execution runs into the infinite loop between two symbolic interpretation locations. In this case, the debugger becomes unresponsive. Or, interpretation is performed only inside this infinite loop. This would correspond to an infinite sequence of interpretation locations. It is assumed in this subsection, that the infinite loop does not contain any decision that depends on symbolic input (such a location would be interpreted and the loop detected with the solver).

This section presents an algorithm for automated detection of infinite loops with dynamic analysis. It is sufficiently lightweight to be applied continuously at runtime, i.e., during concrete execution. It does not rely on a constraint solver. The algorithm is based on *autocorrelation*, a computation commonly used in many areas of applied statistics: One example is in time series analysis to identify periodic changes. Another example is in signal processing, for synchronization in communication systems. Here, autocorrelation is applied as an efficient way to compute the lengths of periodic branch sequences on-the-fly.

Algorithm

Autocorrelation is the correlation of a function $f(n)$ with itself at different points in time. It computes the similarity between the function and a time-lagged version of itself, where the time lag l is variable. The (discrete) autocorrelation R_{ff} at lag l for a real-valued function $f(n)$ is:

$$R_{ff}(l) = \sum_{n\in\mathbb{Z}} f(n)f(n-l)$$

Detecting periodic infinite loops For a periodic function $f(n)$ it is:

$$f(n+p) = f(n)$$

where p is the period length. The autocorrelation function $R_{ff}(l)$ always has a peak value for $l = 0$. For a periodic function of period p, the autocorrelation also peaks at the period length $l = p$ and its integer multiples. For the detection of infinite loops in programs, the autocorrelation is slightly adapted as described in the following. A path through a program can be represented by the corresponding sequence of branches, i.e., the sequence of branch target addresses. The branches comprise both conditional branches and unconditional branches (jumps). A branch target address sequence is denoted as:

$$b(n) \quad n \in [0, m]$$

where $b(0)$ is the first branch after the program's entry point, and $b(m)$ is the last taken branch, leading to the current program execution state. The basic idea is to detect an infinite loop through the branch sequence, which is assumed to become periodic. To this end, the autocorrelation of the branch sequence is computed on-the-fly during program execution.

$$R_{bb}(l, m) = \sum_{n=1}^{m} b(n)b(n-l)\,, \quad l = 0..m$$

The value for $l = 0$ is not of interest and not computed, because an infinite loop has a positive period length. If the program runs into an infinite loop with period length p, the autocorrelation will show a peak at $R_{bb}(l = p, m)$, where the peak value increases with the number of branches m. Unlike in other applications of autocorrelation where a periodic function undergoes some time variance due to noise, here there are have identical periods. Therefore, the multiplication is replaced with Kronecker's delta function:

$$\delta(i, j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{else} \end{cases} \qquad i, j \in \mathbb{Z}$$

That yields:

$$R_{bb}(l, m) = \sum_{n=1}^{m} \delta\Big(b(n), b(n-l)\Big)\,, \quad l = 0..m$$

i.e., the autocorrelation value is only increased if the branch target address is identical to the target address of l branches before. The recursive version for on-the-fly computation is:

$$R_{bb}(l, m) = R_{bb}(l, m-1) + \delta\Big(b(m), b(m-l)\Big)$$

Also the falsification of an infinite loop hypothesis is to be considered. If a periodic sequence is broken, the corresponding correlation values are reset. The recursive computation becomes:

$$R_{bb}(l, m) = \begin{cases} R_{bb}(l, m-1) + 1 & \text{if } b(m) = b(m-l) \\ 0 & \text{else} \end{cases} \tag{4.2}$$

```
int main(int argc, char **argv) {
  unsigned int i, j, k = 0;
  /* Infinite loop: increment i instead of j */
  for (j = 1; j < 0x10000; i++)
    for (i = 0; i < 0x10; i++)
      k++;
}
```

Figure 4.8 – Sample C program containing an infinite loop

```
004004b6 <main>:
004004b6:       push    rbp
004004b7:       mov     rbp,rsp
004004ba:       mov     DWORD PTR [rbp-0x14],edi
004004bd:       mov     QWORD PTR [rbp-0x20],rsi
004004c1:       mov     DWORD PTR [rbp-0x8],0x1
004004c8:       jmp     4004e5 <main+0x2f>
004004ca:       mov     DWORD PTR [rbp-0x4],0x0
004004d1:       jmp     4004db <main+0x25>
004004d3:       add     DWORD PTR [rbp-0xc],0x1
004004d7:       add     DWORD PTR [rbp-0x4],0x1
004004db:       cmp     DWORD PTR [rbp-0x4],0xf
004004df:       jbe     4004d3 <main+0x1d>
004004d7:       add     DWORD PTR [rbp-0x4],0x1
004004e5:       cmp     DWORD PTR [rbp-0x8],0xffff
004004ec:       jbe     4004ca <main+0x14>
004004ee:       mov     eax,0x0
004004f3:       pop     rbp
004004f4:       ret
```

Figure 4.9 – Compiled version of the sample program

An infinite loop candidate is identified if the correlation exceeds a pre-defined threshold value T:

$$R_{bb}(l,m) > T \quad \text{for any } l \tag{4.3}$$

the period of the infinite loop is $p = l$, i.e., the index of the correlation value that first exceeds the threshold T. The location of the infinite loop is given by the p last branch target addresses. The threshold T is configured by the tool user. The choice of T is a trade-off between detection speed and false detection rate, discussed on page 81.

Correlation length In order to limit the buffer length for branch target addresses and autocorrelation values to a constant size independent of the length of a program path, the autocorrelation length can be limited. $R_{bb}(l,m)$ is then only computed for indices $l \in [1..l_{max}]$. With such a fixed correlation length, the algorithm detects infinite loops with a period of up to $p = l_{max}$.

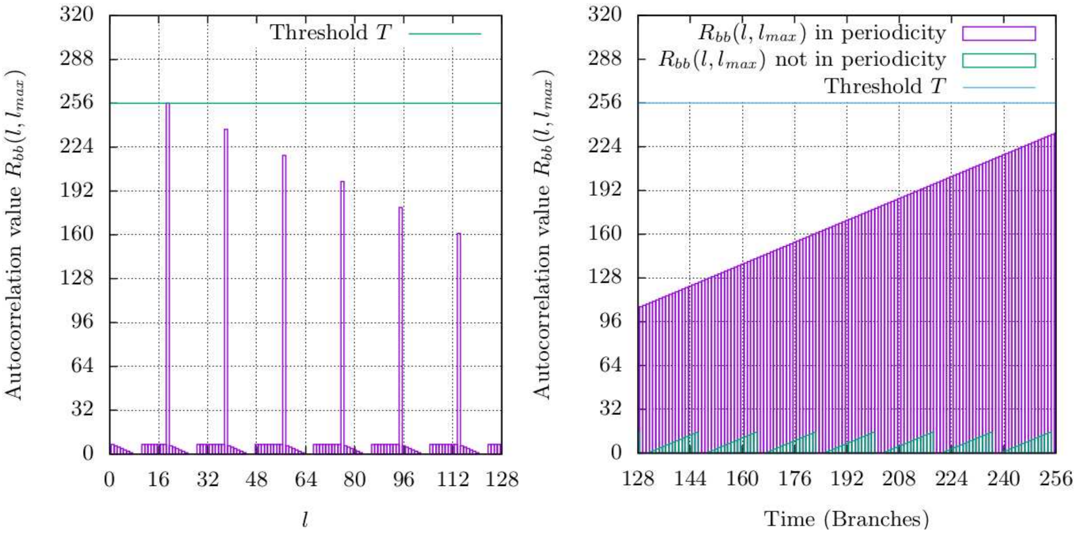


Figure 4.10 – Snapshot of autocorrelation values (left) and variation over time (right) [IKP16]

Illustration For illustration, a sample program is shown in Figure 4.8. The corresponding assembly code is shown in Figure 4.9 (compiled with `gcc`). The outer (infinite) loop is transformed into an unconditional branch at address `0x4004c8` and a conditional one at address `0x4004ec`. Similarly, the branches at `0x4004d1` and `0x4004df` represent the inner loop. Local variables `i, j` and `k` are addressed relatively to the base pointer `rbp` at offsets `-0x4`, `-0x8` and `-0xc` within the current stack frame. Figure 4.10 shows the autocorrelation values at the time of (forced) program termination (that is $\exists l\colon\ R_{bb}(l, m) \geq T$) when choosing $l_{max} = T = 2^8$. The peaks at positions p which are non-zero multiples of 19 clearly indicate the periodicity of the generated assembly code: To perform one complete iteration of the outer loop, the conditional branch at `0x4004df` corresponding to the inner loop is considered 17 times (16 times taken, once not taken) by the algorithm, and with the conditional jump at `0x4004ec` and the unconditional jump at `0x4004d1` this results in a total of 19 branches. The second diagram in Figure 4.10 visualizes the behaviour of two selected autocorrelation values while the instrumented program is executing. One illustrated autocorrelation value is congruent to the periodicity of the sample program. The other illustrated value is its direct neighbour value. Both are plotted over time. The diagram shows that the first value increases steadily while the second autocorrelation value follows a sawtooth-like shape. The diagram illustrates, that any threshold T will eventually be reached in case of a periodic infinite loop with number of branches up to the correlation length. It also shows, that non-infinite loops will not be falsely reported as infinite if the threshold T is big enough.

Detecting certain non-periodic infinite loops The algorithm can be generalized to also detect non-periodic infinite loops, as long as the branch sequences in the loop have equal length. The following

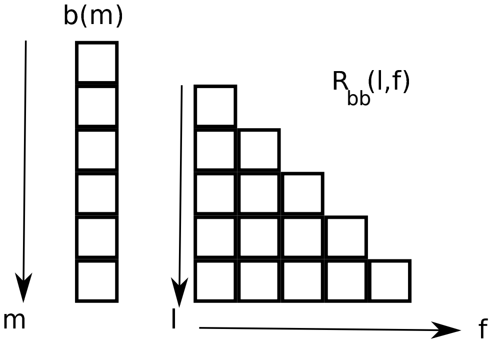


Figure 4.11 – Triangular coefficient matrix

recursive correlation detects re-visited branches with constant branch sequence length:

$$R_{bb}(l,f,m) = \begin{cases} R_{bb}(l,f,m-1) & \text{for } f \not\equiv m \pmod{l} \\ R_{bb}(l,f,m-1)+1 & \text{for } f \equiv m \pmod{l} \wedge b(m) = b(m-l) \\ 0 & \text{for } f \equiv m \pmod{l} \wedge b(m) \neq b(m-l) \end{cases} \tag{4.4}$$

where f is an offset modulo l. The coefficient vector becomes a triangular matrix (compare Figure 4.11). The number of coefficient computations is the same as before, i.e., for every new branch and for each l, only one coefficient is updated (the one for which $f \equiv m \pmod{l}$). But the number of coefficients increases to:

$$N_{\text{coeff}} = \frac{l_{\max}}{2}(l_{\max} - 1),$$

i.e., the space for coefficient storage increases quadratically with correlation length.

Avoiding false positives: SMT-based verification of candidate loops Depending on the threshold value and the application, the presented algorithm has false positive detections. If false positives are not tolerable, there is a possibility for verification or falsification of infinite loop candidates. One iteration of the candidate loop is executed with symbolic execution, and an SMT solver is used to check whether the loop is indeed infinite. This is basically the non-termination check from equation 4.1 from page 66. If the candidate is not verified, the threshold can be increased by some factor. The low-complexity candidate detection by autocorrelation reduces the number of SMT solver queries needed for infinite loop detection, because location and loop length of a candidate are known.

```
#include ”pin.H”                    /* All other includes omitted for readability reasons */
#define T           (500)           /* Threshold triggering an abort */
#define M           (16)            /* Number of values */
#define MASK(X)     ((X) % M)
size_t cur = 0;
unsigned long dst[M] = { 0 };       /* The m last branch targets */
unsigned long cnt[M] = { 0 };       /* The m values R_bb(l, m) */

void Branch(unsigned long ip, bool taken, unsigned long target, unsigned long fallthrough) {
        if (!taken) target = fallthrough;
        /* Check if T has been surpassed by any of the counters and update autocorrelation values */
        for (size_t i = 0; i < M; i++) {
                cnt[i] = (dst[MASK(cur - i)] == target) ? cnt[i] + 1 : 0;
                if (cnt[MASK(cur - i)] >= T)
                        error(-1, 0, ”Infinite␣loop␣detected.␣Instruction:␣%p␣Target:␣%p\n”, ip, target);
        }
        /* Store most recent branch target */
        cur = MASK(cur);
        dst[cur++] = target;
}

void Instruction(INS ins, void *v) {
        /* For any newly translated branch instruction in the main image, add a callback to Branch() */
        if (INS_Category(ins) == XED_CATEGORY_COND_BR || INS_Category(ins) == XED_CATEGORY_UNCOND_BR) {
                IMG img = IMG_FindByAddress(INS_Address(ins));
                if (IMG_Valid(img) && IMG_IsMainExecutable(img))
                        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)Branch, IARG_INST_PTR, IARG_BRANCH_TAKEN,
                                        IARG_BRANCH_TARGET_ADDR, IARG_FALLTHROUGH_ADDR, IARG_END);
        }
}

int main(int argc, char **argv) {
        if (PIN_Init(argc, argv)) return -1;
        INS_AddInstrumentFunction(Instruction, 0);
        PIN_StartProgram();
}
```

Figure 4.12 – Implementation based on the Pin instrumentation framework [IKP16]

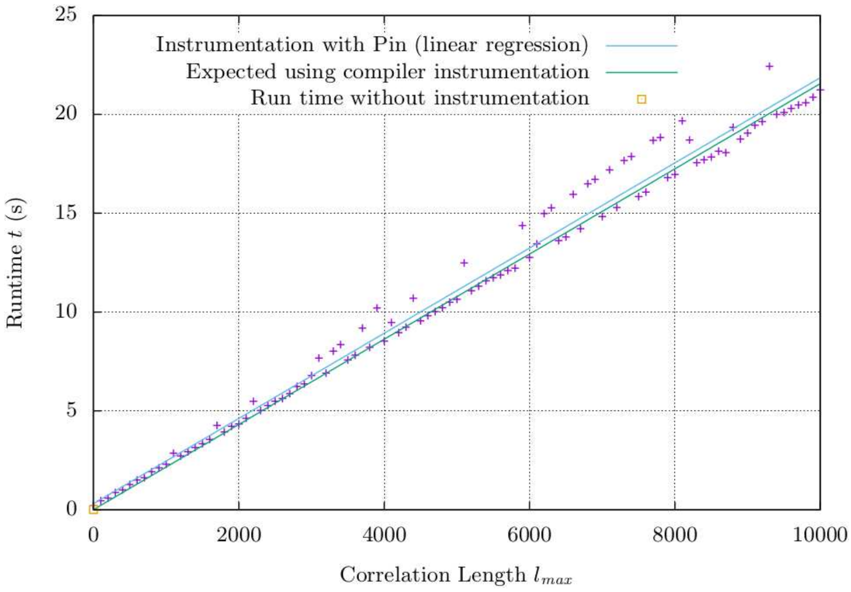


Figure 4.13 – Overhead of autocorrelation based detection of infinite loops [IKP16]

Implementation

The algorithm is implemented based on the dynamic binary instrumentation engine Pin [LCM+05]. Dynamic binary instrumentation provides a mechanism to monitor, inspect and alter the execution of any given binary program at runtime. This is typically achieved by injecting callback functions into a Just-In-Time (JIT) compiled version of the instrumented program which can then observe and/or manipulate the internal state of the program. Dynamic binary instrumentation allows for fine-grained inspection of arbitrary code without needing an executable's source code.

For application in the presented symbolic execution setup, dynamic binary instrumentation can not be used, because it leads to problems with breakpoints. Therefore, implementation must use static binary instrumentation, preferably integrated with the compiler. Nevertheless, dynamic binary instrumentation is useful as first evaluation.

The source code of the implementation is shown in Figure 4.12: The code adds a callback function `Branch()` to all conditional and unconditional branches that reside in the virtual address space of the main executable program image. It is the injected handler function's responsibility to re-compute each autocorrelation value R_{bb} (line 12) and to store the most recent branch target. Moreover, the `Branch()` function ensures that T is still the upper bound of all values residing in R_{bb}. In case of a violation of this last constraint, the Pin tool outputs a warning in line 15.

Properties

Time Complexity The number of operations performed by the algorithm per branch depends on the correlation length. The algorithm computes equations (4.2) and (4.3) for each new branch. As can be derived from the formula for $R_{bb}(l, m)$, the overhead increases linearly with correlation length l_{max}, and is independent of program length. To confirm this, an infinite-loop-free (terminating) version of the sample program is instrumented with the algorithm implementation as Pin tool. The total runtime for different values l_{max} is measured. In the changed sample program, line 6 is updated to increment the local variable `j` instead of `i`. Benchmark results are shown in Figure 4.13.

Space Complexity The amount of memory required to store autocorrelation values and branch target addresses increases linearly with correlation length and is independent of program size.

Runtime Overhead The runtime overhead of the Pin based instrumentation consists of three parts. A one-time overhead independent of program length and a dynamic constant overhead for every branch are due to Pin. The third part is the actual autocorrelation, that causes a constant overhead for every branch, where the overhead increases linearly with correlation length (number of correlation coefficients).

The Intel Pin creators estimate the runtime overhead added by the instrumentation engine at a factor of 2.8 with just the JIT compiler enabled and at an average factor of 7.8 for a basic-block counting instrumentation tool in the worst case [LCM+05]. Figure 4.13 shows a run time of about 2 ms for the modified (terminating) test program running without instrumentation. The same program takes about 200 ms running within an empty Pin instance. Further, a runtime of about 450 ms is obtained for the sample program running within Pin using a correlation length of 100. This implies a slowdown of factor $\frac{450}{2} = 225$ for the example correlation length.

An implementation of the algorithm as compiler instrumentation (static binary instrumentation at compile time) would get rid of the overhead introduced by Pin. The overhead added by Pin are the fixed one-time overhead introduced by Pin's JIT as well as the dynamic part that depends on the number of branches within the target program. Both are independent of the correlation length l_{max}. A compiler based instrumentation therefore would show the same slope in dependence on l_{max}. This runtime is depicted as a dashed (green) line in Figure 4.13, below the blue linear regression curve for measured values with Pin instrumentation. The blue line is the graph of the linear function $t(l_{max}) = 2156.2 \cdot l_{max} + 302700$ (t in milliseconds). For $l_{max} = 100$ the slowdown factor decreases to about 100 by changing from dynamic instrumentation to static instrumentation. Additionally, the example program consists of an unnaturally

Testcase	Runtime (s)
Infinite_Loop__do_01	0.20
Infinite_Loop__do_true_01	0.20
Infinite_Loop__for_01	0.14
Infinite_Loop__for_empty_01	0.19
Infinite_Loop__while_01	0.14
Infinite_Loop__while_true_01	0.14

Table 4.2 – Error detection runtimes for the infinite loop test cases from the Juliet suite

CVE	Name	l_{max}	T	Detection Time	Measured Periodicity
CVE-2011-1027	cgit	64	1024	0.23 s	22
CVE-2011-1002	Avahi	64	1024	0.07 s	3
CVE-2010-4645	PHP	128	1024	0.78 s	69

Table 4.3 – Detection runtimes for selected CVEs [IKP16]

high fraction of branches compared to control-flow preserving instructions of about $\frac{1}{3}$. Benchmarks from [HJ99] show a fraction of branch instructions between 5% and 24% (page 232 in [HJ99]). For real-world programs, one would expect this fraction between 10% and 20%. This would decrease the slowdown factor again, resulting in a slowdown factor of about 50 to 70 for real-world programs with compiler instrumentation and correlation length 100.

Trade-offs The two algorithm parameters l_{max} for correlation length and T as detection threshold are configurable. There are two trade-offs. The first trade-off concerns the number of false negative detections versus algorithm complexity. By increasing correlation length, the number of false negative detections is reduced at the expense of increased program overhead. The second trade-off concerns the number of false positive detections versus detection delay. By increasing the threshold T, the number of false positive detections is reduced at the expense of increased detection delay. An infinite loop must then be executed for more iterations until the threshold is reached. In the practical tests described in the following, the threshold could be set very high (10^6 or more) without causing any significant detection delay.

Experiments

The Pin implementation of the algorithm is tested with the infinite loop test programs from the Juliet suite and with real-world programs.

Juliet suite For infinite loops, the Juliet suite contains 6 test programs. The infinite loops are periodic and do not have a period larger than 2. The terminating 'good' loops are breaked after 10 iterations. Correct detection without false positive therefore requires $l_{max} > 2$ and $T > 10$. The algorithm implementation with Pin is applied to the compiled test programs with parameters $T = 500$ and $l_{max} = 16$. The tool correctly detects the contained infinite loops without false positive and without false negative detections. The runtimes until detection of the respective infinite loop are given in Table 4.2.

Compared to the detection with static symbolic execution as described in Section 4.2.3, the dynamic instrumentation based detection is an order of magnitude faster. For bigger test programs, a larger speed-up with dynamic instrumentation is expected, because on the small Juliet test programs a considerable portion of the runtime is used to perform the instrumentation.

Real-world programs To show that the algorithm is applicable to real-world problems, publicly known infinite-loop vulnerabilities in three widely-used open-source programs are triggered and successfully detected. The three vulnerabilities outlined below have been selected from the CVE list based on the ease of triggering the bug, i.e., it seemed that comparably little effort would be needed to reproduce the bug.

— *cgit* is a web frontend for git repositories written in C.[*] Prior to version 0.8.3.5, cgit contained an infinite loop bug that could be triggered by clients sending an invalid hex escape in the URL query to the remote side. This bug has been assigned CVE-2011-1027.

— *Avahi* is a Unix daemon providing service discovery in local networks which is enabled by default in many popular Linux distributions.[†] Before version 0.6.29, Avahi would enter an infinite loop upon receiving a zero-length UDP packet (CVE-2011-1002).

— *PHP: Hypertext Processor* is a highly popular server-side scripting language for web development.[‡] All versions before 5.2.17/5.3.5 hang infinitely when processing the string representation of the floating-point value $2.2250738585072011 \cdot 10^{-308}$. This infinite loop bug is referenced as CVE-2010-4645.

The results of the evaluation are shown in Table 4.3. It features the detected jump target address period length (as defined by the l value which caused $R_{bb}(m, l)$ to exceed the threshold T) as well as the wall-clock time difference between triggering the bug and forced termination of the instrumented program using the algorithm implementation.

In all three cases, the threshold value $T = 1024$ is used. T was arbitrarily chosen to 1024 to prevent

[*]. http://git.zx2c4.com/cgit/

[†]. http://avahi.org/

[‡]. http://php.net/

false detections with a very small T (it was not tested for which value of T false detections would begin). Tests with larger T up to 10^6 did not show any significant detection slow-down. The tests are initially conducted with $l_{max} = 64$. This yields positive results for the cgit and Avahi tests, detecting the infinite loops of periodicities 22 and 3 within 0.23 and 0.07 seconds, respectively. $l_{max} = 64$ is too small for the PHP test, since that bug exhibits a period length of 69 branches: After increasing l_{max} to 128, the infinite loop is detected within 0.78 seconds.

4.3.4 Information Exposures

This section deals with the detection of information exposure (information leak) errors and the implementation of an information exposure checker. The checker is evaluated with Juliet test programs for information exposures (CWE-526,534,535). Secure information flow and information exposure bugs can be modelled based on a lattice [Den76, DD77]. The model uses trust boundaries, security labels, sanitization and (de)classification functions. Data flows through an application through system calls, i.e., I/O functions. The trust boundary here is the process boundary. Data gets attached (in shadow memory) a security level. Data is allowed to flow from lower to higher security levels. From higher to lower security levels, data flow is restricted unless the data passes through a declassification function (which reduces the security level). Additionally, the information flow can be further restricted in the sense of input filtering. All program input first has to pass through a function, that is pre-defined as sanitization function (input filter). Data flow rules can be either hard-coded in a checker (e.g., based on restrictions and side information inherent in the standard library and standard crypto libraries) or they can be configurable. Security levels can be pre-defined for certain parameters in library calls, e.g., encryption / decryption or I/O functions. Binary instrumentation then performs a security level inference over the data flow on the taken program path. At an output function (crossing the trust boundary with output), it is checked whether the output contains a parameter with too high security level. If so, an information exposure error is reported.

Implementation For a quick evaluation, the security level inference and information exposure checks are not implemented as binary instrumentation, but in the symbolic interpreter (on the source level). Evaluation of the checker therefore uses static symbolic execution, where every CFG node on a path is symbolically interpreted. Offloading the computation to dynamic analysis (e.g., with compiler based instrumentation) would of course enable much faster execution of each path and would enable selective symbolic execution also with this checker.

```
#define N_ITERS 1000000
void CWE366_Race_Condition_Within_Thread__int_byref_12_bad() {
  if(global_returns_t_or_f())     {
    std_thread thread_a = NULL, thread_b = NULL;
    int val = 0;
    if (!std_thread_create(helper_bad, (void*)&val, &thread_a)) {
      thread_a = NULL;
    }
    if (!std_thread_create(helper_bad, (void*)&val, &thread_b)) {
      thread_b = NULL;
    }
    if (thread_a && std_thread_join(thread_a)) std_thread_destroy(thread_a);
    if (thread_b && std_thread_join(thread_b)) std_thread_destroy(thread_b);
    printIntLine(val);
  } else {
    std_thread thread_a = NULL, thread_b = NULL;
    int val = 0;
    if (!std_thread_lock_create(&g_good_lock)) { return; }
    if (!std_thread_create(helper_good, (void*)&val, &thread_a)) {
      thread_a = NULL;
    }
    if (!std_thread_create(helper_good, (void*)&val, &thread_b)) {
      thread_b = NULL;
    }
    if (thread_a && std_thread_join(thread_a)) std_thread_destroy(thread_a);
    if (thread_b && std_thread_join(thread_b)) std_thread_destroy(thread_b);
    std_thread_lock_destroy(g_good_lock);
    printIntLine(val);
  }
}
static void helper_bad(void *args) {
  int *p_val = (int*)args;
  for (int i = 0; i < N_ITERS; i++)     {
    *p_val = *p_val + 1;
  }
}
int global_returns_t_or_f() {
  return (rand() % 2);
}
```

Figure 4.14 – Data race condition; example from Juliet suite [BB12]

Experiments The checker is evaluated with test programs from the Juliet suite for information exposures through environment variables (CWE-526), through debug log files (CWE-534) and through shell error messages (CWE-535). For these common weakness types, Juliet contains 90 test programs. The bugs are correctly detected in all test programs apart from 5 false negative detections, in about 2s per test program. The false negatives are due to missing support of goto statements in the CFG builder version and missing support of data flow through unions in the symbolic interpreter version that were used for the tests [MEI14].

4.3.5 Data Race Conditions

A data race means that there are two concurrent accesses from different threads to the same variable, of which at least one is a write. Data race bugs are introduced in multi-threaded software when the developer forgets to lock a resource, that is shared between threads. Because races are observed only for certain thread interleavings depending on the scheduler's decisions, they are difficult to reproduce and

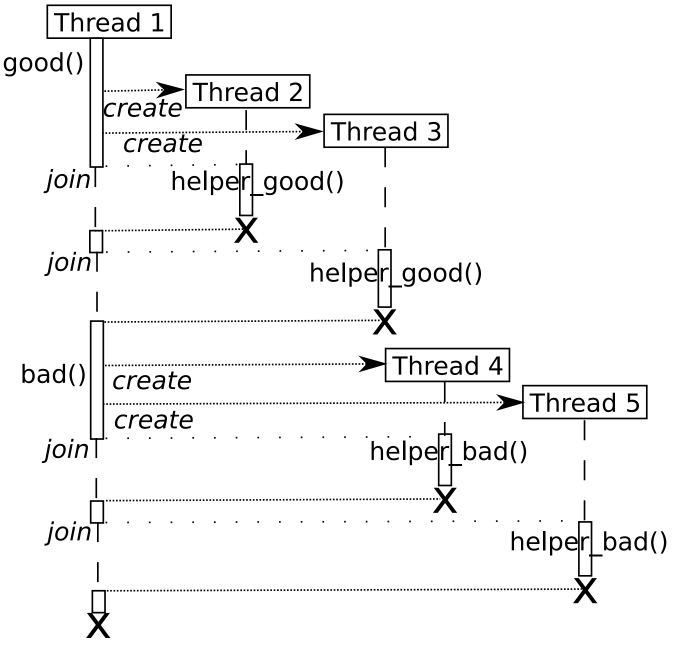


Figure 4.15 – FIFO scheduling on one CPU core [Ibi16b]

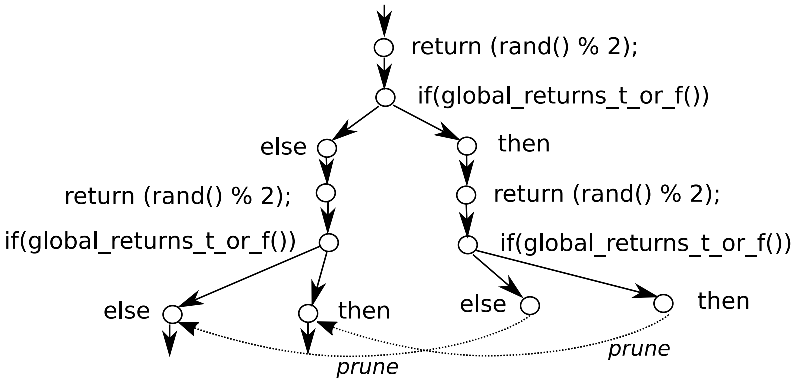


Figure 4.16 – Traversed part of execution tree [Ibi16b]

sometimes called 'Heisenbugs'. Exactly locating feasible data races is known to be NP hard [NM92]. This subsection describes data race detection using dynamic symbolic execution and hybrid lockset / happens-before analysis. Symbolic execution is used to explore the execution tree of multi-threaded software for FIFO scheduling on a single CPU core.

As described in Section 3.1, the analysis of a program path should continue after the detection of a potential data race, in order to achieve code coverage. This means, that actual races should be avoided while still detecting them. Data races can be prevented by using FIFO scheduling on one CPU core. With this scheduling, potential data races can still be detected using happens-before, lockset or hybrid analysis.

The presented approach achieves approximate race detection with both false negative and false positive detections. Concolic execution of multi-threaded code with FIFO scheduling on one CPU core in

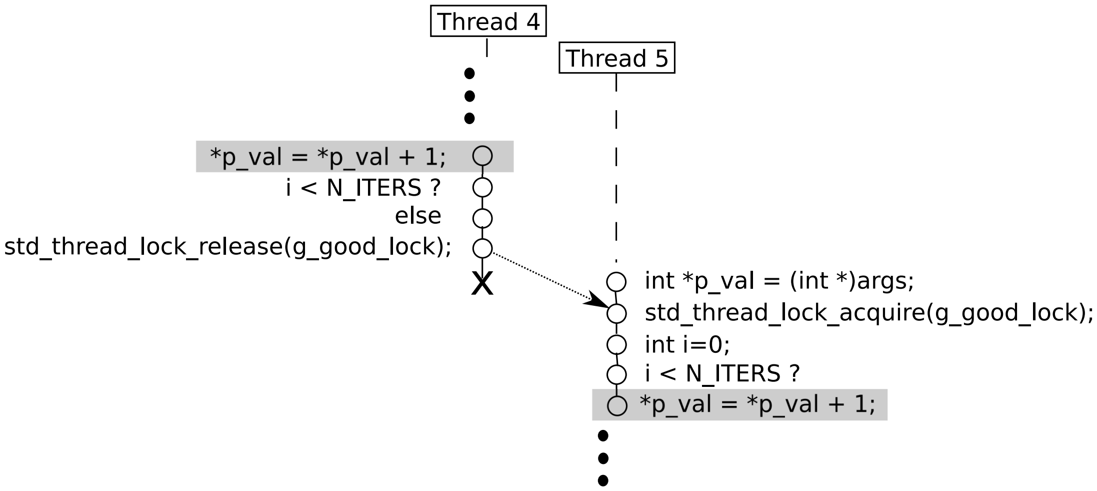

Thread 4
Thread 5
*p_val = *p_val + 1;
i < N_ITERS ?
else
std_thread_lock_release(g_good_lock);
X
int *p_val = (int *)args;
std_thread_lock_acquire(g_good_lock);
int i=0;
i < N_ITERS ?
*p_val = *p_val + 1;

Figure 4.17 – Happens-before analysis [Ibi16b]

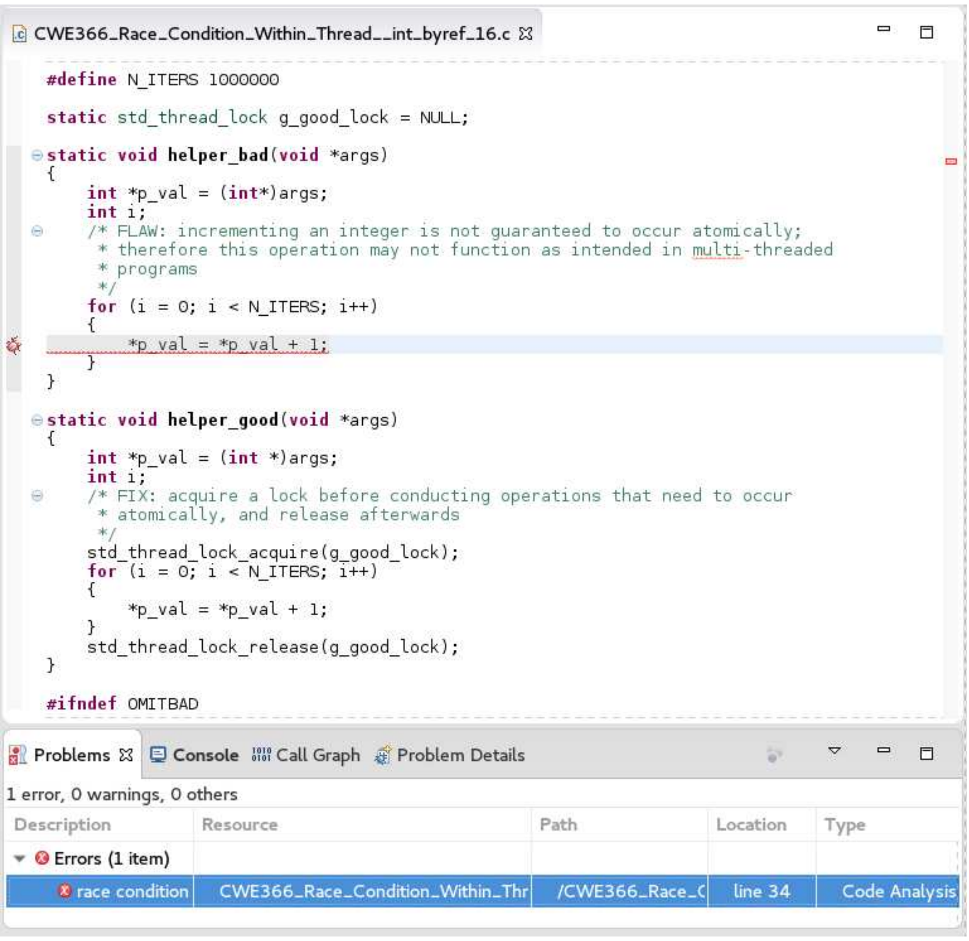
general misses relevant interleavings and therefore leads to false negative race detections. The hybrid happens-before / lockset detection algorithm in general also has false positive detections.

Dynamic race detection using binary instrumentation Race detection on one path works faster during concrete execution. Event tracing and instrumentation do not need a symbolic interpreter. Data races are detected with hybrid dynamic analysis during concrete execution using the available `ThreadSanitizer` algorithm and implementation [SI09]. This paragraph shortly reviews the algorithm. Happens-before analysis with vector clocks is combined with lockset analysis to reduce the number of false negative detections. The hybrid detection has false positive detections. False positives can be eliminated by adding annotations to the program. Binary instrumentation is used to trace the relevant events [SI09]:

- memory access events: *read* and *write*
- synchronization events: locking and happens-before arcs. Locking events are *write lock*, *read lock*, *write unlock* and *read unlock*. Happens-before events are *signal* and *wait*.

The events are traced as state machines in shadow memory. The state machines can be run as pure happens-before or as hybrid analysis, with configurable context tracing information (speed versus information).

Implementation The implementation uses compiler instrumentation with `ThreadSanitizer`, which is featured by `gcc`. The program under test is linked statically with the `ThreadSanitizer` library, and a breakpoint is set on the race detection error report function. If this breakpoint is hit, the stack is traced back to a source file location, where the race is reported. `ThreadSanitizer` supports dynamic annotations (C makros), which can be used if standard Posix threads are not used. They can also be used



```
#define N_ITERS 1000000

static std_thread_lock g_good_lock = NULL;

static void helper_bad(void *args)
{
    int *p_val = (int*)args;
    int i;
    /* FLAW: incrementing an integer is not guaranteed to occur atomically;
     * therefore this operation may not function as intended in multi-threaded
     * programs
     */
    for (i = 0; i < N_ITERS; i++)
    {
        *p_val = *p_val + 1;
    }
}

static void helper_good(void *args)
{
    int *p_val = (int *)args;
    int i;
    /* FIX: acquire a lock before conducting operations that need to occur
     * atomically, and release afterwards
     */
    std_thread_lock_acquire(g_good_lock);
    for (i = 0; i < N_ITERS; i++)
    {
        *p_val = *p_val + 1;
    }
    std_thread_lock_release(g_good_lock);
}

#ifndef OMITBAD
```

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
race condition	CWE366_Race_Condition_Within_Thr	/CWE366_Race_C	line 34	Code Analysis

Figure 4.18 – Screenshot data race detection [Ibi16b]

to eliminate false positive detections and to hide benign races [SI09]. Parts of the code can be marked as safe by the tool user. `ThreadSanitizer` can be run as happens-before or hybrid analysis. Also in pure happens-before mode it can report the involved locks. The slow-down by the instrumentation is reported as factor 20 – 50, and up to several hundred MB can be consumed for shadow memory [SI09].

Experiments The implementation is evaluated with the data race test cases from the Juliet suite [BB12] for common weakness CWE-366 'race condition within a thread'. The test cases are 38 small artificial programs with 5-7 threads each. There are two sets of 19 programs each. One set contains data races on global variables, the other contains data races on stack variables with access through pointers. Both sets cover the same 19 different data and control flow variants, that include conditional branches, loops, goto statements etc. The programs under test are run as unoptimized code with default `ThreadSanitizer` settings without any annotations.

An example 'bad' function is shown in Figure 4.14. It contains a data race on a stack variable with access by reference in line 34. The control flow depends on random input that is returned by a global

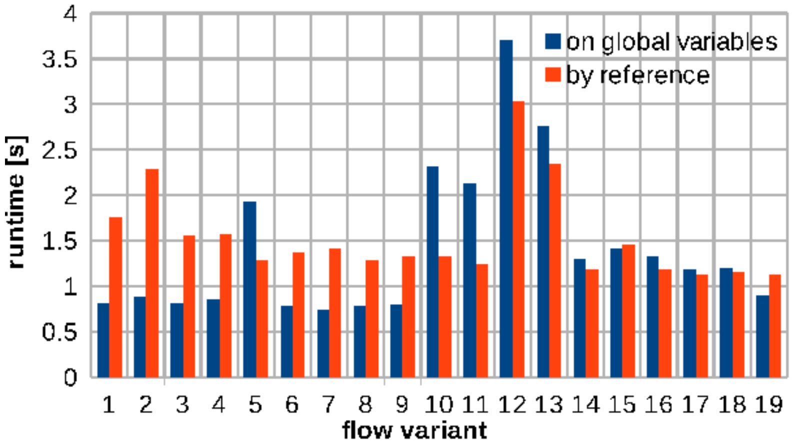


Figure 4.19 – Analysis runtimes for data race tests from Juliet suite [Ibi16b]

function. The `rand()` call in line 38 generates this program input, i.e., the debugger breaks at this call, and the return value of `rand()` is treated as unconstrained symbolic variable. Before this ’bad’ function, the program executes a similar ’good’ function with proper locking in both branches, that also executes a loop with 10^6 iterations. The test program contains many branches, but few of them depend on symbolic input. The loops in the ’good’ and ’bad’ functions are executed concretely. Because they contain only variables and branches depending on concrete input, there is no need for symbolic interpretation. The symbolic execution finds four satisfiable program paths that non-deterministically depend on two `rand()` calls, and of which two paths exhibit the data race. FIFO scheduling on one CPU core for this program is illustrated in Figure 4.15. This figure looks the same for any of the four paths. The part of the execution tree (under this scheduling), that is traversed by symbolic execution, is illustrated in Figure 4.16. It only shows the locations where the debugger stopped. Two paths are followed to program end, the other two are pruned as indicated in the figure. The data race is accurately detected by `ThreadSanitizer`. An example part of the happens-before analysis is illustrated in Figure 4.17, for the `else` branch of the example function, where proper locking is used. The shaded memory access events are separated by a locking arc.

All data races in the 38 test programs are accurately detected without false positives or false negatives. The (wall-clock) analysis runtimes are shown in Figure 4.19. The horizontal axis indicates the Juliet flow variant number. The average analysis runtime is below 2s. Error reporting is shown in Figure 4.18.

4.3.6 Deadlocks and Multiple (Un)Locks

This subsection considers the detection of deadlocks (CWE-833), unlock of a resource that is not locked (CWE-832), multiple locks of a critical resource (CWE-764) and multiple unlocks of a critical resource (CWE-765). A deadlock is a circular locking dependency between threads, where threads wait on each other forever. The described bug types can be detected with state machines by instrumenting and tracing synchronization events, i.e., locking and unlocking operations. This is available as part of the instrumentation necessary for data race detection (Subsection 4.3.5). The available `ThreadSanitizer` also reports these bug types.

4.3.7 Dead Code

Dead code is known under the common weakness enumeration as CWE-561. For dead code, normally a warning is issued rather than an error report. For certain safety critical system, code traceability is required, i.e., every source code line must be mapped to some requirement and test case. In this case, dead code can be seen as an error.

Dead code can be detected with symbolic execution, if at least branch coverage is achieved. Reachability in general is undecidable. For a concrete program it can be decided whether maximum branch coverage was achieved after symbolic execution. For single-threaded code, this is the case under following conditions:

— all input is treated as symbolic

— symbolic execution is run with at least branch coverage (for example simply with path coverage, without any merging)

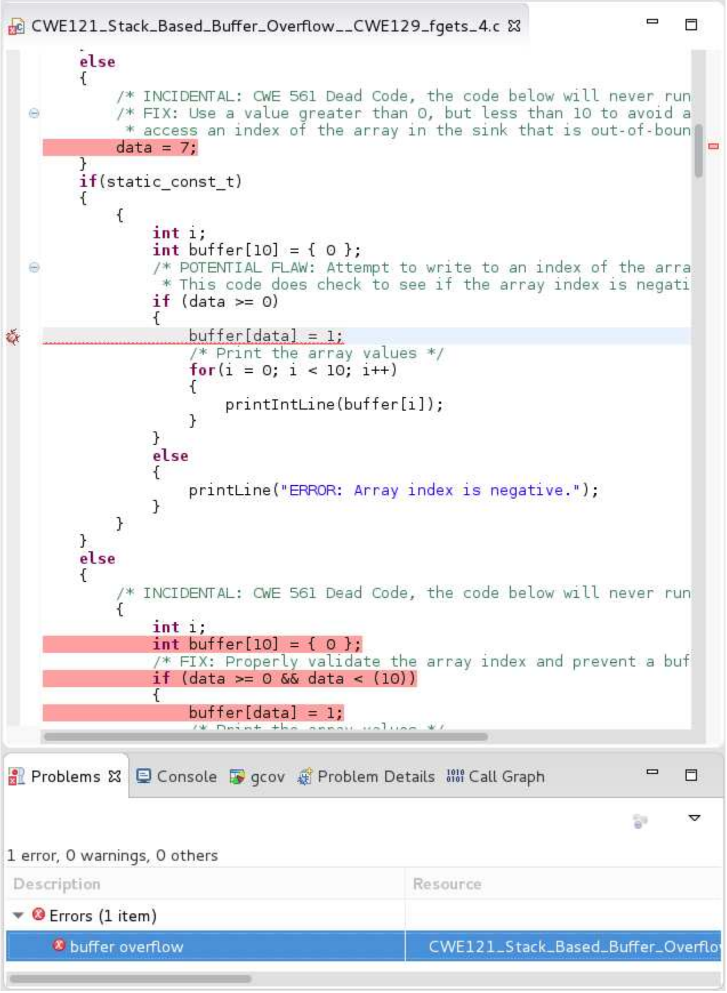
— no concretization has been done

— symbolic execution has terminated, i.e., the execution tree was finite

When it is known that symbolic execution has achieved maximum branch coverage, the remaining uncovered code must be dead.

Implementation The program under analysis is compiled with the available `gcov` instrumentation to trace coverage information (count taken branches). During concrete execution, `gcov` collects the coverage information and writes it into two separate text files (with increments for repeated program executions). It is parsed with CDT's `gcov` parser and can be displayed as editor annotation in the GUI. Uncovered code can be marked in the GUI, as illustrated in Figure 4.20.

Experiments Most test programs of the Juliet suite contain dead code. Here, the same buffer overflow test cases with `fgets` are used as in section 3.2. They cover all of Juliet's flow variants. Figure 3.2 on page 29 shows the number of branch nodes, number of live branch nodes (reachable with at least one satisfiable program path) and number of symbolic branch nodes (decisions depending on symbolic input,



```
else
{
    /* INCIDENTAL: CWE 561 Dead Code, the code below will never run
    /* FIX: Use a value greater than 0, but less than 10 to avoid a
     * access an index of the array in the sink that is out-of-boun
    data = 7;
}
if(static_const_t)
{
    {
        int i;
        int buffer[10] = { 0 };
        /* POTENTIAL FLAW: Attempt to write to an index of the arra
         * This code does check to see if the array index is negati
        if (data >= 0)
        {
            buffer[data] = 1;
            /* Print the array values */
            for(i = 0; i < 10; i++)
            {
                printIntLine(buffer[i]);
            }
        }
        else
        {
            printLine("ERROR: Array index is negative.");
        }
    }
}
else
{
    /* INCIDENTAL: CWE 561 Dead Code, the code below will never run
    {
        int i;
        int buffer[10] = { 0 };
        /* FIX: Properly validate the array index and prevent a buf
        if (data >= 0 && data < (10))
        {
            buffer[data] = 1;
```

1 error, 0 warnings, 0 others

Description	Resource
Errors (1 item)	
buffer overflow	CWE121_Stack_Based_Buffer_Overflo

Figure 4.20 – Error reporting and dead code marking

symbolically interpreted). Dead code is annotated as source code comments in the Juliet test cases. This is used to validate, that dead code is correctly marked. A screenshot is shown in Figure 4.20, dead code lines are shaded in red.

There is a special case of obviously dead source lines. An example is Juliet's flow variant 2, which contains an

```
if (false) {...}
```

statement. Such obviously dead code is detected by the compiler and not translated, even without any code optimization parameters set. Because these source lines do not make it into the binary, they might be called 'non-code'. Correspondingly, also the CFG builder detects these CFG nodes as dead. After querying the CFG builder for dead nodes, these dead non-code source lines can also be marked in the GUI.

Chapter 5

Further Applications

The algorithms described so far, especially interpolation based path merging (Section 3.6), can be useful for further applications.

5.1 Path Merging and Interpolant Weakening in Heuristic Path Exploration

To prove the unreachability of annotated error locations for program verification, path pruning based on unsat-cores of unsatisfiable branches was proposed in [JSV09]. The execution tree traversal order was DFS. In order to apply this path merging approach with a different traversal order, especially with heuristic path exploration, a differentiation between 'half interpolant' (subtree only incompletely traversed) and 'full interpolant' was proposed in [JMN13]. Both publications assume that all potential error locations are annotated with if/else statements.

The algorithm, that achieves 'error and branch coverage' in the work at hand (Section 3.6.3) applies unsat-core based merging in an almost identical way as described in [JSV09]. The work at hand differs in that potential errors are not annotated, and bug conditions are generated automatically during analysis. This shows, that unsat-cores for unsatisfiable branches are something different from unsat-cores for unsatisfiable bug conditions. This difference is exploited in the work at hand for further weakening computed interpolants in case of bug detections (Section 3.6.3).

The algorithm proposed in [JMN13] could benefit from the additional weakening adjustment in the same way. When a bug is detected, then constraints stemming only from the previously unsatisfiable bug condition could be removed from 'half interpolants' and 'full interpolants' (because each bug is only to be detected on one path). This further weakening of interpolants would allow for more path merging and therefore an analysis speedup, without loss of detection accuracy.

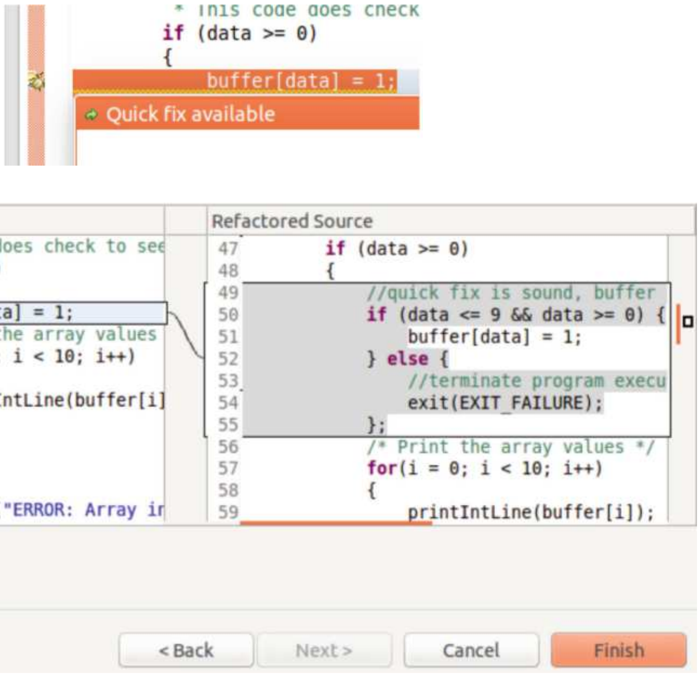
5.2 Interpolation Based Path Merging for Automated Test Case Generation

Automated test case generation is the automated selection of (concrete) program input vectors, so that the execution coverage with these inputs satisfies or approximates a chosen coverage criterion. One approach is to randomly generate test cases and then reduce the set to the required coverage criterion. Symbolic execution can be used for automated test case generation [CGP$^+$06, CDE08]. Symbolic execution builds path constraints, where a path constraint determines the set of input vectors, for which the program takes the respective path. Example input can be generated with the SMT solver's model generation functionality.

The presented coverage-achieving algorithms based on interpolation based path merging can be used to automate the choice of input vectors, that constitute a test suite with desired coverage. Testing normally uses complete program paths, path pruning is not (yet) used in test suite execution. When using the presented algorithms for test case generation, path pruning (program termination) needs to be added when reading from an empty input vector during test suite execution.

Branch Coverage Several symbolic execution tools use heuristic path exploration in order to quickly achieve branch coverage. The most prominent example is `KLEE` [CDE08]. In its current version, it supports path merging based on live variable analysis, but not (yet) any solver-based interpolation. Path merging with unsat-core based interpolation for branch coverage (Section 3.6.4) could be used as a complementary approach.

Modified Condition / Decision Coverage Test cases with MC/DC are typically used for safety-critical software, to show that each condition within a decision independently and correctly affects the decision outcome. In [RH03], an approach for MC/DC test case generation with counter-examples from an LTL model checker is described. A genetic algorithm for test case generation is presented in [AAA09]. An approach to MC/DC system test case generation from MC/DC unit test cases based on symbolic execution is presented in [Pre03]. Symbolic execution and path merging with unsat-core based interpolation for MC/DC (Section 3.6.5) could be used for speed-up or as complementary approach respectively.



```
 * This code does check
if (data >= 0)
{
    buffer[data] = 1;
```

Quick fix available

```
        * This code does check to see
    if (data >= 0)
    {
        buffer[data] = 1;
        /* Print the array values
        for(i = 0; i < 10; i++)
        {
            printIntLine(buffer[i]
        }
    }
    else
    {
        printLine("ERROR: Array ir
```

```
    if (data >= 0)
    {
        //quick fix is sound, buffer
        if (data <= 9 && data >= 0) {
            buffer[data] = 1;
        } else {
            //terminate program execu
            exit(EXIT_FAILURE);
        };
        /* Print the array values */
        for(i = 0; i < 10; i++)
        {
            printIntLine(buffer[i]);
```

Figure 5.1 – Fail-secure quick-fix refactoring [MKIE15]

5.3 Hardware Assisted Infinite Loop Checks at Runtime

In Section 4.3.3, an algorithm for infinite loop detection as dynamic analysis was described and evaluated with binary instrumentation. It detects infinite loop candidates with a modified autocorrelation of the branch sequence.

It is possible to integrate the algorithm in processors, in order to benefit from hardware acceleration [IK16]. Hardware acceleration would eliminate the runtime software overhead at the expense of some extra transistors. Most CPUs already have a branch target address cache. The additional hardware would mainly consist of storage for the correlation coefficients, and the logic to update coefficients and compare with the threshold. Detection of an infinite loop candidate could be signaled with an interrupt to the operating system and with an operating system signal to a process. The operating system could perform verification / falsification with an SMT solver query after unrolling the loop once, to eliminate false positives. Only one SMT query is needed, because the location and length of the infinite loop candidate are detected by correlation.

5.4 Automated Quick-Fix Generation for Detected Bugs

Information from error detection can be used for automated program transformation (AST transformation). The transformation can be a fail-secure quick-fix, or in some cases an error correction.

Fail-secure error mitigation With 'fail-secure' (unlike fail-safe) it is meant here, that the exploitation of a bug is prevented. Set constraints for input that triggers the bug can be determined from symbolic execution [BNS+08]. In principle, it is possible to insert one or more branches into a program before the bug location, so that execution is stopped with an error report for program input that would trigger the bug. This is easy, e.g., when the size of a buffer that might overflow is context-insensitive.

Fail-secure quick-fix generation for buffer overflow bugs is tested with 54 test programs for stack-based buffer overflows from the Juliet suite, that were already used in chapter 4.3.7. Figure 5.1 shows on the top an example quick-fix context menu. On the bottom, it shows an example for a proposed refactoring.

Error correction One application of automated error detection is automated correction of the detected errors. For certain common weaknesses, automated correction is possible. An example are locking errors [FF05]. In case of a detected data race, lock and unlock operations can be automatically inserted. For double lock or double unlock errors, similar changes of synchronization can be inferred. Automated error correction benefits from faster or more accurate error detection and comes closer to practical applicability.

5.5 Automated Input Filter Generation

Another application of automated error detection is automated generation of an input filter. Equivalent to in-place fail-secure quick-fixes, the bug-triggering input vectors can be prevented from being read into the program. One approach is to drop network input in a firewall. Such an input filter is preferable, because the computation does not have to be stopped. Computing weakest precondition of a bug's path constraints and generating regular expression signatures is described in [BNS+08]. Automated input filter generation benefits from faster or more accurate error detection and comes closer to practical applicability.

5.6 Path Merging in Symbolic Execution Based Automated De-Obfuscation

Automated reversing or de-obfuscation is used, e.g., for malware analysis. Symbolic execution on the binary level can be used for this purpose [YD15]. Symbolic execution of binaries means symbolic CPU simulation. Reversing requires reconstruction of control flow graphs on the binary level. When instruction addresses are traced, explored program paths can be folded into CFGs. Obfuscated code often uses self-modification, e.g., code can be XORed with some data in a loop before execution. In this case, the control flow graph becomes context sensitive. Memory writes and the execution of overwritten code can be detected during symbolic execution. From this, a version tree (containing all context-sensitive code variants) of the binary could be built. Automated de-obfuscation could benefit from the presented path merging for branch coverage (Section 3.6.4), because CFG reconstruction only requires branch coverage.

5.7 Path Merging in Retargetable Symbolic Execution

The standard approach for symbolic execution of machine code is to lift machine code instructions to intermediate code, and perform symbolic execution on the intermediate level. There are different frameworks for symbolic execution of intermediate code (e.g., [BJAS11, SBY$^+$08]). Intermediate code has the advantage of a small instruction set and is independent of any special CPU.

Architecture description languages (ADL) are used for the design of application-specific instruction set processors [SML07]. Syntax and semantics of CPU instructions are specified in the architecture description language. The effect of changes in the instruction set on application performance can be quickly evaluated. From the architecture description, compiler backend, instruction set simulator, debugger and binary utilities can be automatically generated. An example ADL is ArchC [ARB$^+$05, BCR$^+$08], which is an extension of SystemC. Symbolic CPU simulation can be based on an ADL. This has the conceptual advantage, that the instruction semantics is specified only once, and not separately in CPU, instruction set simulator and each analysis tool or code lifter. From the ADL, either a lifter to an intermediate language or directly a symbolic execution engine could be generated. This is illustrated in Figure 5.2.

Retargetable symbolic execution, i.e., generation of a symbolic execution engine for machine code based on an ADL, could benefit from path merging in the same way as symbolic execution on any other code level.

5.8 Path Merging in Symbolic Execution Based WCET Determination

For hard real-time applications, it is necessary to determine the worst-case execution time (WCET) [WEE$^+$08]. This is essentially the longest (satisfiable) path problem. WCET can be computed with symbolic execution [LS99]. This can be implemented either as symbolic execution on the binary level, or by back-annotating basic blocks with instruction counts.

Also WCET analysis with symbolic execution could benefit from a speed-up through path merging.

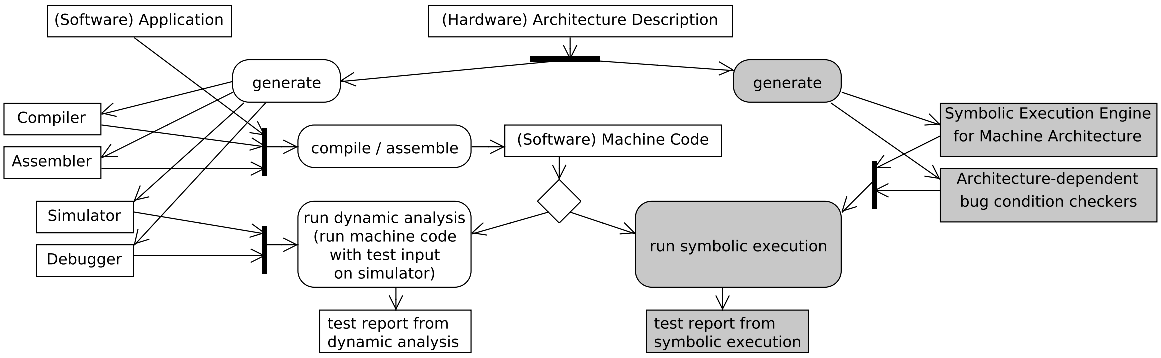

(Software) Application
(Hardware) Architecture Description
generate
generate
Compiler
Assembler
compile / assemble
(Software) Machine Code
Symbolic Execution Engine for Machine Architecture
Architecture-dependent bug condition checkers
Simulator
Debugger
run dynamic analysis (run machine code with test input on simulator)
run symbolic execution
test report from dynamic analysis
test report from symbolic execution

Figure 5.2 – Tool flow based on architecture description [Ibi15a]

While interpolation with unsat-cores does not seem applicable, path merging based on live variable analysis seems feasible.

5.9 Interpolation Based Merging in Selective Symbolic Execution for Remote Debugging

It is often desirable to separate symbolic execution from the host machine into a virtual machine (VM), or to perform concolic execution in a remote target system with special peripheral hardware. S2E [CKC11] is a symbolic execution tool, that analyzes binaries in a virtual machine. It is implemented using the qemu virtual machine with an LLVM backend, on which the KLEE [CDE08] symbolic execution engine is applied. S2E employs the worklist algorithm with qemu snapshots. It uses 'lazy concretization' and describes different relaxed consistency models for (return) variables, and their effect on the set of analyzed paths and false negative / false positive bug detections. Avatar [ZBFB14] makes S2E usable with physical target hardware (to avoid time consuming writing of special qemu hardware models). It uses instrumentation in a target, and redirects hardware I/O to concolic execution in parallel in S2E. Avatar uses gdb both to control the target device through a JTAG (Joint Test Access Group) interface gdb server and to control S2E through the gdb server in qemu.

Neither S2E nor Avatar use any path merging. Concolic execution in a virtual machine or in a remote target could benefit from path merging just like the other described applications.

Chapter 6

Related Work

Symbolic Execution There is a large body of work on symbolic execution available, which spans over 30 years. Several survey articles are available [CS13, CSG+11, PV09, ABC+13]. Prominent dynamic symbolic execution tools include `DART` [GKS05], `CUTE` [SMA05], `EXE` [CGP+06], `SAGE` [GLM08], `KLEE` [CDE08] and `Mayhem` [CARB12, ARCB14]. `EXE` and `KLEE` use heuristic breadth-first execution tree traversal with the worklist algorithm. Selective symbolic execution is presented with `S2E` [CKC11]. It allows to choose which parts of a system are executed symbolically. It uses the `qemu` virtual machine monitor with an LLVM backend and runs the engine from `KLEE` on it.

There are several approaches that are closely related to automated bug detection with symbolic execution. One approach is annotation-based verification, which proves the absence of errors. The annotations reduce the context that is necessary for analysis. An annotation language for C is `ACSL` [BCF+13], one for Java is `JML` [BCC+05]. Prominent verification tools for C are `Frama-C` [CCK+15] and `VCC` [CDH+09]. Another approach is symbolic model checking [CGP99], where the whole program is treated as a formula. Bounded model checking for C is implemented in `CBMC` [CKL04]. An approach that offers a smooth transition between static analysis and verification is extended static checking [DLNS98].

The work at hand differs from previous work on symbolic execution in that is uses selective symbolic execution of multi-threaded code with interpolation based path merging and combines it with binary instrumentation.

Different levels of software Symbolic execution has been applied on the software architecture level to models [PPW+05], e.g., to UML-RT state diagrams [ZD12]. On the source code level, symbolic execution has been applied to a variety of languages. Examples for C are [RSM+10, CGP+06]. Most tools perform symbolic execution on an intermediate code representation. `KLEE` runs on LLVM intermediate code. Pex [TH08] performs dynamic symbolic execution of the Common Intermediate Language (MSIL/CIL). The engine described in [VPK04] analyzes Java bytecode. Binary code has been analyzed by lifting to an intermediate representation and symbolic execution of the intermediate code, described in [SBY$^+$08, BJAS11]. Analysis of x86 binaries with symbolic execution is implemented in SAGE [GLM08]. The work at hand differs in that it uses selective symbolic execution on the C source code level through a debugger machine interface.

Bounded Search In order to limit the size of the traversed scheduling and execution tree, bounds can be applied, e.g., to the number context switches, loop unrollings or path lengths [QR05, RG05, CFMS09, CKL04]. KLEE requires the tool user to specify input bounds. The work at hand uses the more recent approach of path merging to mitigate the path explosion problem.

Coverage Heuristics Breadth-first, depth-first and structural heuristics are used in [APV07, GV02]. More heuristics are presented in [BH09]: a 'look-ahead' heuristic to stop path exploration when no uncovered location can be reached, a 'max-calldepth' heuristic to avoid backtracking from deep function calls. and a 'solve-first' heuristic to explore shorter path prefixes early. In SAGE, a limited generational search is started from pre-defined input. An approach with evolutionary testing (genetic algorithm) is described in [IX08]. In [MC12], the coverage is based on an existing test suite. An approach to path exploration based on length-n subpath program spectra is presented in [LSWL13]. A predictive path filtering heuristic to skip paths that don't improve coverage is discussed in [SPF$^+$14], test case reduction and prioritization is discussed in [ZGA14]. The work at hand uses the more recent approach of path merging rather than heuristics to mitigate the path explosion problem.

Parallelization Memory and computation requirements of symbolic execution motivated a distributed version of KLEE, presented in [BUZC11]. SAGE performs symbolic execution of one path at a time in parallel on different machines. From a path trace, it generates input vectors for related paths (by switching a decision). Parallelized symbolic execution of Java Bytecode with a coverage heuristic is described in [SP10]. In order to exploit the full path merging potential, the work at hand uses depth-first exploration without parallelization.

Partial Order Reduction Another way of pruning the tree is partial order reduction [CGMP99, KWG09, FG05, AAJS14], which prunes away irrelevant thread interleavings. The number of traversed states of the scheduling and execution tree is reduced, while maintaining complete coverage of behaviour, e.g., for the detection of races and deadlocks. Dynamic partial order reduction [FG05] traces the happens-before relation for thread interactions to find backtracking points for branching. Optimal dynamic partial order reduction [AAJS14] explores a minimum number of representative thread interleavings. The work at hand differs in that it sacrifices data race detection accuracy for a speed-up by analyzing fewer interleavings. It rather uses selective symbolic execution of multi-threaded code with FIFO scheduling on one CPU core, interpolation based path merging and race detection with binary instrumentation.

Abstraction and Path Merging Other related work uses abstraction, i.e., generalization of constraints, to merge more paths. Abstract interpretation [CC77] allows for complete bug detection (no false negatives), but introduces false positives (unsound). An approach to automatically generate an abstraction based on predicates over decision conditions contained in the program source is presented in [GS97]. Counter-example guided abstraction refinement [CGL+03] is an automated abstraction refinement to iteratively undo unsound path merges in order to remove false positives. A coverage based abstraction with refinement is proposed in [BBDP10]. It uses iterative search for satisfiable paths to a program location. Another method is presented in [JSV09] and further developed in [McM10, JNS11, JMNS12]. It uses logic interpolation [Cra57] and weakest precondition computing during backtracking of error-free paths, so that further error-free paths explored later could be merged. In order to alleviate the path explosion problem, it is shown in [BCE08] that a live variable analysis can be applied so that program paths, that only differ in dead variables, can be merged. The implementation extends `EXE`. Merging of paths with live differences is investigated in [KKBC12, ARCB14]. Path disjunctions are used in the corresponding logic formulation passed to the solver. Heuristics for path merging are presented, which aim at balancing computational effort between the symbolic execution frontend and the SMT solver backend. Path merging based on interpolation using unsat-cores is described in [JSV09]. The latter approach is more comprehensive than merging based on live variable analysis. It comprises elimination of constraints for dead variables, because those are not present in backtracked formulas. The interpolation based merging approach is used in the static symbolic execution tool `TRACER` [JMNS12] for verification. It has been combined with heuristic path exploration [JMN13] and with partial order reduction for multi-threaded code [CJ14]. In [BE13], an approach of path merging during depth-first traversal is described, that does not use an SMT solver. The removal of constraints from the path constraint is based on dynamic slicing and dependence analysis. Path merging is decided by checking the inclusion of constraint sets. The implementation is based on `KLEE` [CDE08]. The work at hand shows the relation of several known approaches as part of a coverage and interpolation hierarchy. It presents a symbolic execution algorithm with interpolation based path merging to achieve branch coverage. Further, a new path merging refinement is presented that exploits interpolant weakening in case of error detections.

Detection of Infinite Loops relies on a threshold for a maximum duration of unresponsiveness (used, e.g., in DART), on automated theorem proving [GRH+08, VR08, PS09, BSOG11, CCF+14, BJSS09], or on comparing program states at branches [CMKR11]. The prominent tools use static symbolic execution of the program source code, with loop invariant generation [GRH+08, VR08] and/or checking for satisfiable fixed-points or more generally recurrence sets [GRH+08, CCF+14]. Proving non-termination is sound, i.e., it does not yield any false positive infinite loop detections. In Looper [BJSS09], it is proposed to use a theorem prover to verify that a program is stuck in an infinite loop. The tool is to be run at user's request, and single-steps the unresponsive program with symbolic execution. In [CMKR11], it is proposed to apply program state comparisons only on demand in case a process becomes unresponsive. As noted in [CMKR11], this approach misses infinite loops where program states are not identical. Comparing process states as described in [CMKR11] is also not sound, i.e., it might false positively report an infinite loop, because there is also a kernel state (packet queues etc.) for a running process. The kernel state is not included in the state comparisons. The work at hand presents a new approach for infinite loop detection based on modified autocorrelation.

Detection of Data Race Conditions Dynamic detection at runtime is a practical way for race detection, that does not need a constraint solver. One technique is to instrument memory accesses and thread interaction with binary instrumentation and check for races using the happens-before relation [Lam78] with vector-clocks. Happens-before analysis may have false negative detections depending on the scheduling. It is more sophisticated than lockset analysis, but scales worse with an increasing number of threads. LiteRace applies sampling, i.e., it monitors only a subset of all memory accesses. It can detect a majority of races by monitoring a small number of accesses [MMN09]. FastTrack is an optimized implementation of happens-before analysis with reduced complexity [FF09]. Pacer [BCM10] combines sampling with FastTrack. DataCollider [EMBO10] implements memory access sampling with hardware breakpoints and watchpoints and applies it to kernel code. Lockset analysis is used in Eraser [SBN+97]. It instruments memory accesses and traces thread locksets and variable locksets. If a variable access is not protected by a lock, then a warning is issued. Lockset analysis is lightweight and scales well. On the downside, it leads to more false positive detections than happens-before analysis. Hybrid race detection is presented in [OC03] as a two-pass solution. First, locksets are used to find problematic variables. Then, happens-before analysis is applied only to those variables. In [ISM99], the DJIT algorithm is presented, which is a variation of happens-before analysis. MultiRace [PS03] combines DJIT with locksets to reduce false positives. A location's lockset is reset at synchronization barriers. ThreadSanitizer [SI09] applies static binary instrumentation for happens-before and lockset analysis and is integrated with several current C compilers. Dynamic race detection is integrated in the managed runtime environments RaceTrack [YRC05] and Goldilocks [EQT10]. In [ZTZ07, DWS$^+$12], it is proposed to integrate hardware acceleration for race detection into CPUs.

Static Analysis offers a possibility to detect races without false negatives. A method that requires programmer annotations and is based on type inference is described in [AFF06]. The tools RELAY [VJL07] and LOCKSMITH [PFH11] apply lockset analysis statically with data flow analysis. According to Palsberg [EP14], "the best existing static technique" is implemented in Chord [Nai08], but it "reports a large number of false positives that would be daunting to examine by hand".

The prominent symbolic execution tools DART, CUTE and KLEE currently do not feature race detection. Symbolic execution tools that do support race detection are jCUTE [SA06], Con2colic [FHRV13] and LCT [KSH13]. They use a solver to search paths through the program's joint execution and scheduling tree, i.e., the solver determines both program input and thread scheduling. The combinatorial explosion can be partly mitigated with partial order reduction [FG05]. jCute [SA06] determines program input and thread schedule with the solver to explore different paths and interleavings, and it detects races when they occur. Con2colic also determines input and schedule with the solver. It implements a heuristic to first achieve branch coverage, and then explore an increasing number of context switches. Also Con2colic can detect a race when it occurs (no happens-before or lockset analysis). LCT [KSH13] implements concolic execution with dynamic partial order reduction. In [EP14], the tool Racageddon is described. It starts with race candidates that have been found with an existing hybrid technique. It then uses concolic execution to search for input and schedule that lead to a real race (to remove false positives). WHOOP [DDR15] considers races between pairs of entry points to driver code and runs a symbolic lockset analysis with SMT solver. A combination of partial order reduction with interpolation based path pruning is described in [CJ14].

The work at hand presents a new race detection algorithm that uses concolic execution of multi-threaded code with FIFO scheduling on one CPU core, interpolation based path merging and a combination with binary instrumentation for happens-before / lockset analysis.

Chapter 7

Discussion

While most common weaknesses can in principle be automatically detected, practical automated bug detection is a complexity / accuracy trade-off, that is determined by the currently known algorithms. Symbolic execution allows for accurate bug detection, but a naive straight-forward implementation faces the path explosion problem. Currently, the predominant tools use coverage-oriented heuristic path exploration to find bugs under complexity constraints.

The work at hand follows the more recent approach of state interpolation and path merging. It develops algorithms for interpolation based path merging, where the chosen interpolation determines the achieved coverage. This yields a trade-off between computational effort and detection accuracy, that is qualitatively described as coverage and interpolation hierarchy. The trade-off is also quantitatively described for the special case of a certain subset of the Juliet test suite. The evaluation uses on the one hand the implementation-independent measures of false detection rates and path set sizes, and on the other hand the comparison of runtimes of the implementations within the same framework. The fastest analysis mode is a branch coverage heuristic. For more accurate analysis, interpolation based path merging is used, where the coverage is determined by the selected interpolation. For the most accurate analysis mode of error and branch coverage, the merge conditions are updated in case of a bug detection for improved efficiency. The presented approach is debugger based selective symbolic interpretation on the source code level. Source modification is not needed.

The approach is also applicable to symbolic execution in a virtual machine and to remote concolic execution in a physical target system. One application example is to use the `gdb` server in the `qemu` VM. Another application example is to use a `gdbserver` in a target system. In-circuit emulators and JTAG based hardware debuggers also often support the `gdb` machine interface.

One limitation of the presented implementation is that only a small part of the standard library is modelled, just enough to correctly detect the errors for all of the Juliet suite's data and control flow variants, for a small number of Juliet's 'baseline bugs'. Another limitation is slow execution speed. While lots of computation is saved due to path merging, the execution of a single path remains slow. The work at hand uses dynamic symbolic execution on the source code level with selective symbolic tree-based interpretation. An alternative is symbolic interpretation of intermediate code [CDE08], which benefits from a small instruction set. The implementation would be a stack-based or register-based symbolic bytecode interpreter, rather than a tree-based symbolic interpreter. Bytecode interpreters in general are much faster than tree-based interpreters: while tree-based interpretation is 'simple and natural', it is also considered the 'slowest approach' [WWS$^+$12]. The AST contains syntax nodes without related semantic actions, whose traversal is overhead. Method dispatch for each node is costly. Speed differences for concrete interpreters are quantified in [EG03]. A slow (concrete) interpreter can be more than a factor of 1000 slower than native code execution. A fast (concrete) interpreter is only about a factor of 10 slower than native code execution [EG03]. With the debugger-based approach using selective symbolic execution on the one hand execution between breakpoints proceeds with full concrete execution speed. On the other hand, the context switches and communication interfaces between debugger, debugger target, symbolic interpreter and solver processes impose a high overhead. A further limitation is that the work at hand only considers finite execution trees, i.e., programs with only a finite number of satisfiable program paths. Completeness is neither claimed nor considered. Another limitation is that the results are not compared to state of the art symbolic execution engines.

To sum up, the presented implementation is an initial algorithm evaluation. A faster implementation seems possible by implementing interpolation based path merging within a symbolic bytecode interpreter (e.g., [CDE08]) or possibly by using binary instrumentation and compiling everything into one binary. For the future it seems likely that more symbolic execution based bug detection tools will use interpolation based path merging. The adoption of path merging to other applications of symbolic execution like automated de-obfuscation or worst-case execution time determination (merging based on live variable analysis) also seems likely. There are tools that mix heuristic path exploration with path merging. KLEE [CDE08] features path merging based on live variable analysis. In [JMN13], path merging during heuristic path exploration is based on 'half interpolants'. But, in order to exploit the full complexity reduction potential of unsat-core based interpolation, depth-first traversal is required.

Abbreviations

ADL	(Processor) Architecture Description Language
AST	Abstract Syntax Tree
BFS	Breadth First Search
CDT	(Eclipse) C/C++ Development Tools
CFG	Control Flow Graph
CWE	Common Weakness Enumeration
DFS	Depth First Search
DSF	(Eclipse) Debugger Services Framework
DOM	Document Object Model
gdb	GNU Debugger
gcc	GNU Compiler Collection
GUI	Graphical User Interface
IDE	Integrated Development Environment
I/O	Input / Output
JTAG	Joint Test Action Group
LLVM	LLVM (formerly ’Low Level Virtual Machine’)
MC/DC	Modified Condition / Decision Coverage
OS	Operating System
PET	Program Execution Tree
POSIX	Portable Operating System Interface
SAT	Satisfiability
SMT	Satisfiability Modulo Theories
WCET	Worst Case Execution Time
VM	Virtual Machine

Publications

- *"Efficient Data-Race Detection with Dynamic Symbolic Execution"*, A. Ibing; IEEE Software Engineering Workshop 2016
- *"Autocorrelation Based Detection of Infinite Loops at Runtime"*, A. Ibing, J. Kirsch and L. Panny; IEEE Int. Conf. Dependable, Autonomic and Secure Computing 2016
- *"Dynamic Symbolic Execution with Interpolation Based Path Merging"*, A. Ibing; Int. Conf. Advances and Trends in Software Engineering 2016
- *"Dynamic Symbolic Execution using Eclipse CDT"*, A. Ibing; Int. Conf. Software Eng. Advances 2015
- *"Symbolic Execution Based Automated Static Bug Detection for Eclipse CDT"*, A. Ibing; Int. J. Advances in Software, vol. 8, no. 1&2 2015
- *"Architecture Description Language Based Retargetable Symbolic Execution"*, A. Ibing; Design, Automation & Test in Europe 2015
- *"A Fixed-Point Algorithm for Automated Static Detection of Infinite Loops"*, A. Ibing and A. Mai; IEEE Int. Symp. High Assurance Systems Engineering 2015
- *"Automated Detection of Information Flow Errors in UML State Charts and C Code"*, P. Muntean, A. Rabbi, A. Ibing and C. Eckert; IEEE Int. Workshop Model-Based Verification & Validation 2015
- *"Automated Generation of Buffer Overflow Quick Fixes using Symbolic Execution and SMT"*, P. Muntean, V. Kommanapalli, A. Ibing, and C. Eckert; Int. Conf. Computer Safety, Reliability and Security 2015
- *"SMT-Constrained Symbolic Execution Engine for Integer Overflow Detection in C Code"*, P. Muntean, M. Rahman, A. Ibing and C. Eckert; Int. Information Security South Africa Conf. 2015
- *"Path-Sensitive Race Detection with Partial Order Reduced Symbolic Execution"*, A. Ibing; Workshop on Formal Methods in the Development of Software 2014
- *"A Backtracking Symbolic Execution Engine with Sound Path Merging"*, A. Ibing; Int. Conf. Emerging Security Information, Systems and Technologies 2014
- *"Context-Sensitive Detection of Information Exposure Bugs with Symbolic Execution"*, P. Muntean, C. Eckert and A. Ibing; Int. Workshop Innovative Software Development Methodologies and Practices 2014

— *"Parallel SMT-Constrained Symbolic Execution for Eclipse CDT/Codan"*, A. Ibing; Int. Conf. Testing Software & Systems 2013

— *"SMT-Constrained Symbolic Execution for Eclipse CDT/Codan"*, A. Ibing; Workshop on Formal Methods in the Development of Software 2013

— *"Clustering Algorithms for Non-profiled Single-Execution Attacks on Exponentiations"*, J. Heyszl, A. Ibing, S. Mangard, F. Santis and Georg Sigl; Smart Card Research & Advanced Application Conf. 2013

— *"On Implementing Trusted Boot for Embedded Systems"*, M. Khalid, C. Rolfes and A. Ibing.; IEEE Int. Symp. Hardware-Oriented Security and Trust 2013

— *"A Secure Architecture for Smart Meter Systems"*, K. Boettinger, D. Angermeier, A. Ibing, D. Schuster, F. Stumpf and D. Wacker; IEEE Int. Symp. Cyberspace Safety and Security 2012

Patent Application

— *”Prozessor mit korrelationsbasierter Endlosschleifenerkennung”*, German Patent Application 10 2016 113 968.8, 28.07.2016, Applicant: TU München, Inventors: A. Ibing and J. Kirsch

Bibliography

[AAA09] Z. Awedikian, K. Ayari, and G. Antoniol. MC/DC automatic test input data generation. In *Annual Conference on Genetic and Evolutionary Computing*, 2009.

[AAJS14] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *ACM Symposium on Principles of Programming Languages*, 2014.

[ABC+13] S. Anand, E. Burke, T. Chen, J. Clark, M. Cohen, W. Grieskamp, M. Harman, M. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[AFF06] M. Abadi, C. Flanagan, and S. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Programming Languages and Systems*, 28(2):207–255, 2006.

[APV07] S. Anand, C. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java Pathfinder. In *TACAS*, pages 134–138, 2007.

[ARB+05] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros. The ArchC architecture description language and tools. *Int. J. Parallel Programming*, 33(5), 2005.

[ARCB14] T. Avgerinos, A. Rebert, S. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Int. Conf. Software Eng.*, 2014.

[BB12] T. Boland and P. Black. Juliet 1.1 C/C++ and Java test suite. *IEEE Computer*, 45(10):88–90, 2012.

[BBDP10] M Baluda, P. Braione, G. Denaro, and M. Pezze. Structural coverage of feasible code. In *AST*, 2010.

[BCC+05] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[BCE08] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 351–366, 2008.

[BCF+13] P. Boudin, P. Cuoq, J. Filliatre, C. Marche, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C specification language, version 1.9, 2013. retrieved: 05/2015.

[BCM10] M. Bond, K. Coons, and K. McKinley. PACER: proportional detection of data races. In *ACM Conf. Programming Language Design and Implementation*, 2010.

[BCR+08] A. Baldassin, P. Centoducatte, S. Rigo, D. Casarotto, L. Santos, M. Schultz, and O. Furtade. An open-source binary utility generator. *ACM Trans. Design Automation of Electronic Systems*, 13(2), 2008.

[BE13] S. Bugrara and D. Engler. Redundant state detection for dynamic symbolic execution. In *USENIX Annual Technical Conf.*, 2013.

[BH09] S. Bardin and P. Herrmann. Pruning the search space in path-based test generation. In *Int. Conf. Software Testing, Verification and Validation*, pages 240–249, 2009.

[BJAS11] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. BAP: A binary analysis platform. In *Int. Conf. Computer Aided Verification*, pages 463–469, 2011.

[BJSS09] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *Int. Conf. Automated Software Engineering*, 2009.

[BK08] C. Baier and J. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[BNS+08] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Theory and techniques for automated generation of vulnerability-based signatures. *IEEE Trans. Dependable and Secure Systems*, pages 224–241, 2008.

[Bro11] L. Brouwer. Über Abbildungen von Mannigfaltigkeiten. *Mathematische Annalen*, (71), 1911.

[BS08] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Int. Conf. Automated Software Engineering*, pages 443–446, 2008.

[BSOG11] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated detection of non-termination and NullPointerExceptions for Java Bytecode. In *Int. Conf. Formal Verification of Object-Oriented Software*, pages 123–141, 2011.

[BST10] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard – version 2.0. In *Int. Workshop Satisfiability Modulo Theories*, 2010.

[BUZC11] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys*, 2011.

[CARB12] S. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on binary code. In *IEEE Symp. Security and Privacy*, 2012.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified latttice model for static analysis of programs by construction or approximation of fixed points. In *Symp. Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[CCF⁺14] H. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. O'Hearn. Proving nontermination via safety. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 156–171, 2014.

[CCK⁺15] L. Correnson, P. Cuoq, F. Kirchner, V. Prevosto, A. Puccetti, J. Signoles, and B. Yakubowski. Frama-C user manual, release sodium, 2015. retrieved: 05/2015.

[CDE08] C. Cadar, D. Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[CDH⁺09] E. Cohen, M. Dahlweid, M. Hillebrandt, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Int. Conf. Theorem Proving in Higher Order Logics*, pages 23–42, 2009.

[CFMS09] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *Int. Conf. Automated Software Eng.*, 2009.

[CGL⁺03] E. Clarke, O. Grumberg, Y. Lu, S. Jha, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGMP99] E. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *Int. J. Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CGP⁺06] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *13th ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, 2006.

[CJ14] D. Chu and J. Jaffar. A framework to synergize partial order reduction with state interpolation. In *Hardware and Software: Verification and Testing*, pages 171–187, 2014.

[CKC11] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2011.

[CKL04] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[CMKR11] M. Carbin, S. Misailovic, M. Kling, and M. Rinard. Detecting and escaping infinite loops with Jolt. In *European Conf. Object-Oriented Programming*, pages 609–633, 2011.

[Cra57] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[CS13] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[CSG$^+$11] C. Cadar, K. Sen, P. Godefroid, N. Tillmann, S. Khurshid, W. Visser, and C. Pasareanu. Symbolic execution for software testing in practice – preliminary assessment. In *Int. Conf. Software Eng.*, pages 1066–1071, 2011.

[dB08] L. deMoura and N. Bjorner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[dB11] L. deMoura and N. Bjorner. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[DD77] D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[DDR15] P. Deligiannis, A. Donaldson, and Z. Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *Int. Conf. Automated Software Eng.*, 2015.

[DE82] R. Dannenberg and G. Ernst. Formal program verification using symbolic execution. *IEEE Trans. Software Eng.*, 8(1):43–52, 1982.

[Dec03] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Den76] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[DLL62] M. Davis, G. Logeman, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7), 1962.

[DLNS98] D. Detlefs, K. Leino, G. Nelson, and J. Saxe. Extended static checking. SRC Research report 159, Compaq Systems Research Center, 1998.

[DWS$^+$12] J. Devietti, B. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: always-on sound and complete race detection in software and hardware. In *Int. Symp. Computer Architecture*, pages 202–212, 2012.

[EG03] A. Ertl and D. Grepp. The structure and performance of efficient interpreters. *J. Instruction-Level Parallelism*, 5:1–25, 2003.

[EMBO10] J. Erickson, M. Musuvathi, S. Burchhardt, and K. Olynyik. Effective data-race detection for the kernel. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[EP14] M. Eslamimehr and J. Palsberg. Race directed scheduling of concurrent programs. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 301–314, 2014.

[EQT10] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race-aware Jave runtime. *Communications of the ACM*, 53(11):85–92, 2010.

[FF05] C. Flanagan and S. Freund. Automatic synchronization correction. In *Synchronization and Concurrency in Object-Oriented Languages*, 2005.

[FF09] C. Flanagan and S. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, 2009.

[FG05] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM Symposium on Principles of Programming Languages*, pages 110–121, 2005.

[FHRV13] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *ESEC/FSE Joint Meeting on Foundations of Software Engineering*, pages 37–47, 2013.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[GLM08] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp. (NDSS)*, pages 151–166, 2008.

[GRH$^+$08] A. Gupta, A. Rybalchenko, T. Henzinger, R. Xu, and R. Majumdar. Proving non-termination. In *Symp. Principles of Programming Languages (POPL)*, 2008.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Int. Conf. Computer Aided Verification (CAV)*, pages 72–83, 1997.

[GV02] A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *ISSTA*, pages 12–21, 2002.

[HCS06] L. Habets, P. Collins, and J. Schuppen. Reachability and control synthesis for piecewise-affine hybrid systems on simplices. *IEEE Trans. Automatic Control*, 51(6), 2006.

[HJ99] M. Hill and N. Jouppi. *Readings in Computer Architecture*. Morgan Kaufman, 1999.

[Ibi13a] A. Ibing. Parallel SMT-constrained symbolic execution for Eclipse CDT/Codan. In *Int. Conf. Testing Software & Systems*, 2013.

[Ibi13b] A. Ibing. SMT-constrained symbolic execution for Eclipse CDT/Codan. In *Workshop on Formal Methods in the Development of Software*, 2013.

[Ibi15a] A. Ibing. Architecture description language based retargetable symbolic execution. In *Design, Automation & Test in Europe*, 2015.

[Ibi15b] A. Ibing. Dynamic symbolic execution using Eclipse CDT. In *Int. Conf. Software Eng. Advances*, 2015.

[Ibi15c] A. Ibing. Symbolic execution based automated static bug detection for Eclipse CDT. *Int. J. Advances in Security*, 1&2:48–59, 2015.

[Ibi16a] A. Ibing. Dynamic symbolic execution with interpolation based path merging. In *Int. Conf. Advances and Trends in Software Engineering*, 2016.

[Ibi16b] A. Ibing. Efficient data-race detection with dynamic symbolic execution. In *IEEE Software Engineering Workshop*, 2016.

[IK16] A. Ibing and J. Kirsch. Prozessor mit korrelationsbasierter Endlosschleifenerkennung. German Patent Application 10 2016 113 968.8; applicant: TU München, July 2016.

[IKP16] A. Ibing, J. Kirsch, and L. Panny. Autocorrelation based detection of infinite loops at runtime. In *IEEE Int. Conf. Dependable, Autonomic and Secure Computing*, 2016.

[IM15] A. Ibing and A. Mai. A fixed-point algorithm for automated static detection of infinite loops. In *IEEE Int. Symp. High Assurance Systems Engineering*, 2015.

[ISM99] A. Itzkovitz, A. Schuster, and O. Mordehai. Towards integration of data race detection in DSM systems. *J. Parallel and Distributed Computing*, 59:180–203, 1999.

[IX08] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE*, pages 297–306, 2008.

[JMN13] J. Jaffar, V. Murali, and J. Navas. Boosting concolic testing via interpolation. In *Foundations of Software Engineering*, 2013.

[JMNS12] J. Jaffar, V. Murali, J. Navas, and A. Santosa. TRACER: A symbolic execution tool for verification. In *Int. Conf. Computer Aided Verification (CAV)*, pages 758–766, 2012.

[JNS11] J. Jaffar, J. Navas, and A. Santosa. Unbounded symbolic execution for program verification. In *Int. Conf. Runtime Verification*, pages 396–411, 2011.

[JSV09] J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *Int. Conf. Principles and Practice of Constraint Programming (CP)*, pages 454–469, 2009.

[Kin76] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KKBC12] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Conf. Programming Language Design and Implementation (PLDI)*, pages 193–204, 2012.

[KSH13] K. Kähkönen, O. Saarikivi, and K. Heljanko. LCT: A parallel distributed testing tool for multithreaded Java programs. *Electronic Notes in Theoretical Computer Science*, pages 253—259, 2013.

[KWG09] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Int. Conf. Computer Aided Verification*, 2009.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Las15] E. Laskavaia. Codan- a C/C++ code analysis framework for CDT. In *EclipseCon*, 2015.

[LCM+05] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[LJZ07] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *ECEC/FSE*, 2007.

[LP10] D. LeBerre and A. Parrain. The SAT4J library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 7:59–64, 2010.

[LS99] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17:183–207, 1999.

[LSWL13] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *Int. Conf. Object Oriented Programming Systems, Languages and Applications*, pages 19–32, 2013.

[MBC07] R. Martin, S. Barnum, and S. Christey. Being explicit about security weaknesses. *CrossTalk The Journal of Defense Software Engineering*, 20:4–8, 3 2007.

[MC12] P. Marinescu and C. Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *ICSE*, 2012.

[McM10] K. McMillan. Lazy annotation for program testing and verification. In *Int. Conf. Computer Aided Verification (CAV)*, pages 104–118, 2010.

[MEI14] P. Muntean, C. Eckert, and A. Ibing. Context-sensitive detection of information exposure bugs with symbolic execution. In *Int. Workshop Innovative Software Development Methodologies and Practices*, 2014.

[MKIE15] P. Muntean, V. Kommanapalli, A. Ibing, and C. Eckert. Automated generation of buffer overflow quick-fixes using symbolic execution and SMT. In *Int. Conf. Computer Safety, Reliability and Security*, 2015.

[MMN09] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *ACM Conf. Programming Language Design and Implementation*, 2009.

[MW06] A. Muehlenfeld and F. Wotawa. Fault detection in multi-threaded C++ server applications. In *Int. Workshop Multithreading in Hardware and Software*, 2006.

[Nai08] M. Naik. *Effective static race detection for Java*. PhD thesis, Stanford University, 2008.

[NCH$^+$05] G. Necula, J. Condit, M. Harpen, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Trans. Programming Languages and Systems*, 27(3):477–526, 2005.

[NM92] R. Netzer and B. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, pages 74–88, 1992.

[NNH10] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2010.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[NS07] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Int. Conf. Programming Language Design and Implementation*, 2007.

[OC03] R. O'Callahan and J. Choi. Hybrid dynamic data race detection. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2003.

[Par10] T. Parr. *Language Implementation Patterns*. Pragmatic Bookshelf, 2010.

[PFH11] P. Pratikakis, J. Foster, and M. Hicks. LOCKSMITH: Practical static race detection for C. *ACM Trans. Programming Languages and Systems*, 33, 2011.

[PPW$^+$05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Int. Conf. Software Eng.*, pages 392–401, 2005.

[Pre03] A. Pretschner. Compositional generation of MC/DC integration test suites. In *Int. Workshop Test and Analysis of Component-Based Systems*, pages 1–11, 2003.

[PS03] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Int. Parallel and Distributed Processing Symposium*, 2003.

[PS09] E. Payet and F. Spoto. Experiments with non-termination analysis for Java Bytecode. In *BYTECODE*, pages 83–96, 2009.

[PV09] C. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Software Tools Technology Transfer*, 11:339–353, 2009.

[PWCR08] P. Piech, T. Williams, F. Chouinard, and R. Rohrbach. Implementing a debugger using the DSF framework. In *EclipseCon*, 2008.

[QR05] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.

[RG05] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *Int. Conf. Computer Aided Verification (CAV)*, 2005.

[RH03] S. Rayadurgam and M. Heimdahl. Generating MC/DC adequate test sequences through model checking. In *IEEE/NASA Software Engineering Workshop*, 2003.

[RSM+10] E. Reisner, C. Song, K. Ma, J. Foster, and A. Porter. Using symbolic evaluation to understand behaviour in configurable software systems. In *Int. Conf. Software Eng.*, pages 445–454, 2010.

[SA06] K. Sen and G. Agha. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In *Int. Conf. Computer Aided Verification*, pages 419–423, 2006.

[SBN+97] S.Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Trans. Computer Systems*, 15(4):391–411, 1997.

[SBPV12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 28–28, 2012.

[SBY+08] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Int. Conf. Information Systems Security*, pages 1–25, 2008.

[SI09] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, pages 62–71, 2009.

[SMA05] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[SML07] O. Schliebusch, H. Meyr, and R. Leupers. *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer, 2007.

[SN05] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, 2005.

[SP10] M. Staats and C. Pasareanu. Parallel symbolic execution for structural test generation. In *Int. Symp. Software Testing and Analysis*, pages 183–193, 2010.

[SPF$^+$14] T. Su, G. Pu, B. Fang, J. He, J. Yan, S. Jiang, and J. Zhao. Automated coverage-driven test data generation using dynamic symbolic execution. In *Int. Conf. Software Security and Reliability*, pages 98–107, 2014.

[SPS11] R. Stallman, R. Pesch, and S. Shebs. Debugging with gdb, 2011. [retrieved: Sept., 2015].

[TH08] N. Tillmann and J. Halleux. Pex – white box test generation for .NET. In *Int. Conf. Tests and Proofs (TAP)*, pages 134–153, 2008.

[Tur37] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.

[Uni13] United States National Security Agency, Center for Assured Software. *Juliet Test Suite v1.2 for C/C++*, May 2013. [retrieved: Sept., 2015].

[VJL07] J. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *ACM Symp. Foundations of Software Engineering (ESEC-FSE)*, 2007.

[VPK04] W. Visser, C. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Int. Symp. Software Testing and Analysis (ISSTA)*, pages 97–107, 2004.

[VR08] H. Velroyen and P. Rummer. Non-termination checking for imperative programs. In *Tests and Proofs (TAP)*, 2008.

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Pusout, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem — overview of methods and survey of tools. *ACM Trans. Embedded Computing Systems*, 7(3), 2008.

[WWS+12] T. Würthinger, A. Wöss, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Dynamic Languages Symposium*, 2012.

[XTHS09] T. Xie, N. Tillmann, J. Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Int. Conf. Dependable Systems and Networks*, pages 359–368, 2009.

[YD15] B. Yadegari and S. Debray. Symbolic execution of obfuscated code. In *ACM Conf. Computer and Communications Security*, 2015.

[YRC05] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *ACM Operating Systems Review*, 2005.

[ZBFB14] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded system's firmwares. In *Network and Distributed Systems Security Symposium*, 2014.

[ZD12] K. Zurowska and J. Dingel. Symbolic execution of UML-RT state machines. In *ACM Symp. Applied Computing*, pages 1292–1299, 2012.

[ZGA14] C. Zhang, A. Groce, and M. Alipour. Using test case reduction and prioritization to improve symbolic execution. In *Int. Symp. Software Testing and Analysis*, pages 160–170, 2014.

[ZMMM01] L. Zhang, C. Madigan, M. Maskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *IEEE/ACM Int. Conf. Computer-Aided Design*, pages 279–285, 2001.

[ZTZ07] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *Int. Symp. High-Performance Computer Architecture*, 2007.