Symbolic Execution Based Automated Static Bug Detection for Eclipse CDT

Andreas Ibing

Chair for IT Security
TU München, Germany
Email: andreas.ibing@tum.de

***Abstract*—Software vulnerabilities may be exploited for intruding into a system by an attacker. One approach to mitigation is to automatically analyze software source code in order to find and remove software bugs before release. A method for context-sensitive static bug detection is symbolic execution. This article presents an SMT-constrained static symbolic execution engine with sound path merging. The engine is used by checkers for memory access violation, infinite loops, and atomicity violations. Context information provided by the engine is shared by the different checkers. Further checkers can easily be connected. The engine integrates as plug-in extension into Eclipse CDT and uses CDT's parser, AST visitor and CFG builder, as well as Eclipse's GUI and marker framework for bug reporting. The presented approach is evaluated with test cases from the Juliet test suite for C/C++. The evaluation shows a significant speed-up by path merging already for the small Juliet programs. The speed-up depends on the number of decision nodes with more than one satisfiable branch and increases for larger programs.**

Keywords–Static analysis; Symbolic execution.

I. Introduction

This article is an extended version of [1], which presents a backtracking symbolic execution engine with sound path merging on the C source level. This extended version gives a more detailed description, provides more context information and evaluates the symbolic execution engine on a larger test set. Software weaknesses are classified by the common weakness enumeration [2]. For brevity, a software weakness is called bug in this article. If a weakness could be exploited by an attacker, it is a vulnerability. The likelihood of exploit varies for different bug types. For buffer overflows, e.g., the likelihood of exploit is very high [2].

Symbolic execution [3] is a program analysis method, where software input is regarded as variables (symbolic values). It is used to automatically explore different paths through software, and to compute path constraints as logical equations from the operations with the symbolic input. An automatic theorem prover (constraint solver) is used to check program paths for satisfiability and to check bug conditions for satisfiability. The current state of automatic theorem provers are Satisfiability Modulo Theories (SMT) solvers [4], the standard interface is the SMTlib [5]. An example state-of-the art solver is described in [6].

Automatic analysis tools that rely on symbolic execution have been developed for the source-code level, intermediate code and binaries (machine code). Available tools mostly analyze intermediate code, which exploits a small instruction set and certain independence of programming language and target processor. A prominent example is [7], which analyzes Low Level Virtual Machine (LLVM [8]) code. Symbolic execution on the source-code level is also interesting for several reasons. An intermediate representation loses source information by discarding high-level types and the compiler lowers language constructs and makes assumptions about the evaluation order. However, rich source and type information is needed to explain discovered bugs to the user [9] or to generate quick-fix proposals.

In order to detect bugs as early as possible, bug detection tools should be integrated into IDEs. The integration of bug finding tools into IDEs is further important for ease of use and for the integration of different tools. A synergy lies for example in the automated generation of quick-fix refactoring proposals based on detected bug information.

During symbolic execution, a symbolic execution engine builds and analyzes satisfiable paths through programs, where paths are lists of control flow graph (CFG) nodes. Always restarting symbolic execution from the program entry point for different, partly overlapping program paths (path replay) is obviously inefficient. The standard approach is therefore the worklist algorithm [10]. In this algorithm, a list of symbolic program states (the worklist) is kept in memory. These states are the frontier nodes (unexplored nodes) of the program execution tree. While the list is not empty, one symbolic program state is taken from the list and interpreted to yield its successor state(s), which are then added to the list. At program branches, there may be more than one satisfiable successor state. In this case the respective predecessor is cloned before interpretation. The reuse of intermediate analysis results with state cloning has the downside of being memory-intensive. In [11], state cloning with a recursive data structure to store only state differences is used. Another approach for engine implementation is symbolic state backtracking [12]. It keeps only the symbolic program states along the currently analyzed program path in memory (stored incrementally with single assignments) and avoids the inefficiency of path replay as well as the exponential memory consumption of state cloning.

The tree of satisfiable program paths, called the program execution tree, grows exponentially with the number of decisions in the program for which two or more branches are satisfiable. Straight-forward application of symbolic execution is therefore not scalable. This is often called the path explosion problem. In [13], it is noted that program paths can be merged when the path constraints differ only in dead variables, because further path extension would have the same consequences for the paths. It presents an implementation that extends [11]. This implementation uses a cache of observed symbolic program states and introduces a type of live variables analysis, which

it calls read-write-set (RWSet) analysis.

Interesting properties of bug detection algorithms are soundness (no false positive detections) and completeness (no false negatives). Because a bug checker cannot be sound and complete and have bounded runtime, in practice bug checkers are evalutated with measurement of false positive and false negative detections and corresponding runtimes on a sufficiently large bug test suite. The currently most comprehensive C/C++ bug test suite for static analyzers is the Juliet suite [14]. Among other common software weaknesses, it contains test cases for buffer overflows, infinite loops and race conditions. In order to systematically measure false positives and false negatives, it contains both 'good' and 'bad' functions, where 'bad' functions contain a bug. It further combines 'baseline' bugs with different data and control flow variants to cover the language's grammar constructs and to test the context depth of the analysis.

This paper develops and evaluates a sound path merging method in a source-level backtracking symbolic execution engine. The aim is to context-sensitively find bugs in unannotated C code, in the sense of automated testing without test-suite, while alleviating the path explosion problem. The approach is targeted at all C bug types that can be detected as constraint violations. The implementation extends [12]. According to the TIOBE index [15], C is currently the most popular programming language (based on average ranking during the last 12 months). The remainder of this paper is organized as follows. Section II gives an overview of related work. Section III shortly reviews symbolic execution. Section IV describes the tool architecture and design decisions. The description includes the integration in the Eclipse C/C++ development tools (CDT). Section V depicts different checker classes connected to the symbolic execution engine. These checkers are described in more detail in previous publications [12], [16], [17], [18]. Section VI presents results of experiments with test cases from the Juliet suite, with focus on the analysis speed-up provided by path merging. Section VII then discusses the presented approach based on the results.

II. Related Work

There is a large body of work on symbolic execution available, which spans over 30 years [19]. Therefore, only a small selection is named here. More in-depth information is available in survey articles [19], [20], [21], [22].

a) Related approaches: 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 presented in [23], one for Java in [24]. Prominent verification tools for C are described in [25], [26]. Another approach is symbolic model checking [27], where the whole program is treated as a formula. Bounded model checking for C is described in [28]. An approach that offers a smooth transition between static analysis and verification is extended static checking [29]. Symbolic execution can be seen as an instance of abstract interpretation, because some variables have formulas as values, which abstracts from concrete interpretation. Abstract interpretation in a narrower sense [30] is referred to in the paragraph on abstraction.

b) Different levels of software: Symbolic execution has been applied on the software architecture level to models, e.g., to UML-RT state diagrams [31]. Further related state-based work on the model level is presented in [32], [33]. On the source code level, symbolic execution has been applied to a variety of languages. Examples for C are [34], [11]. Most tools perform symbolic execution on an intermediate code representation. Apart from [7], where LLVM intermediate code is analyzed, prominent symbolic execution engines are presented in [35] and [36]. In [35], dynamic symbolic execution of the Common Intermediate Language (MSIL/CIL) is performed. The engine described in [36] analyzes Java bytecode. Binary code has been analyzed by lifting to an intermediate representation and symbolic execution of the intermediate code, described in [37], [38]. Analysis of x86 binaries with symbolic execution is presented in [39].

c) Static and dynamic: Symbolic execution can be applied both as static and as dynamic analysis. The latter is also referred to as concolic testing [40]. Dynamic symbolic execution for test case generation is described in [7], [11], [35], [39], [40], [41]. To reduce complexity and increase analysis speed, as many variables as possible are regarded as concrete values. Normally, only input variables or variables that directly depend on program input are modelled as symbolic. The analysis runs dynamically as long as all parameters are concrete, and equation systems for the solver are smaller. In [40], [41], dynamic symbolic execution is applied on the C source code level. In [39], dynamic symbolic execution is applied for the analysis of x86 machine code. A combination of static and dynamic symbolic execution called selective symbolic execution is presented in [42]. The approach is to specify a system part of interest where symbolic variables are used, and to execute other parts with concrete/concretized values. It uses a hypervisor with LLVM backend and utilizes the symbolic execution engine described in [7].

d) Path merging: Sound path merging based on dead path differences is presented in [13], the implementation extends [11]. Merging of paths with live differences is investigated in [43]. 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. The implementation extends [7].

e) Abstraction: Other related work uses abstraction, i.e., generalization of constraints, to merge more paths. Abstract interpretation [30] 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 [44]. Counter-example guided abstraction refinement [45] is an automated abstraction refinement to iteratively undo unsound path merges in order to remove false positives. Another method is presented in [46] and further developed in [47], [48], [49]. It uses logic interpolation [50] and weakest precondition computing during backtracking of error-free paths, so that further error-free paths explored later could be merged.

f) IDE integration: This paragraph considers only the widely used open-source Eclipse IDE. It is possible to connect external tools as processes to the IDE. There are, e.g., Eclipse

plug-ins available that communicate over character streams with the analysis engines presented in [25], [28]. Eclipse is designed according to the OSGi architecture (formerly known as Open Services Gateway initiative) [51], i.e., it consists of a small runtime, and functionality is provided as plug-ins. It contains its own dependency and update management. Therefore, tight IDE integration offers several advantages, the most important one being the synergies between plug-ins. Eclipse CDT features a code analysis framework [52]. Eclipse also includes a SAT solver [53]. On the other hand, it currently (version 4.4) does not feature an SMT solver, and CDT's code analysis framework does not include path-sensitive or context-sensitive analyses.

III. Symbolic Execution

Symbolic execution can be seen as a state transition system (e.g., [48], [54]). This section shortly describes interprocedural symbolic execution for whole-program analysis.

A. Symbolic program state

A symbolic program state is a quadruple $\sigma = (l, s, \Pi, T)$. l is a program location. On the source code level this means a control flow node. To this program location corresponds a syntax subtree $a(l)$ of the abstract syntax tree (AST) of a source file. s denotes the set of symbolic program variables, i.e., logic equations over symbolic input variables. Π is the path constraint, and T the function call stack.

B. Successor locations

Depending on the type of l, there are the following cases for the set of successor locations of a symbolic program state:

1) $a(l)$ contains one or more not yet evaluated function call expressions. Then the successor location is the start node of the function whose function call expression is next in traversal order of $a(l)$. Location l is saved as return node: $T.push(l)$.
2) l is an exit node and $a(l)$ does not contain an unevaluated call expression. The successor location is the return node from the stack: $l' = T.pop()$. In case of exit from `main()`, symbolic execution terminates.
3) l is *not* an exit node and $a(l)$ does not contain an unevaluated call expression. Then, the set of successor locations are l's children along the edges in the function's control flow graph.

C. Symbolic successor states

For the transition from l to successor l' there are the following cases:

1) l' is a start node. Then $\sigma' = (l', s \wedge c_C, \Pi, T')$, where $c_C = [a_C(l)]$ is the evaluation of the subtree of $a(l)$ rooted in the respective function call expression.
2) l' is a branch node. Then $\sigma' = (l', s, \Pi \wedge c, T')$, where the constraint $c = [a(l)]$ is the evaluation of $a(l)$. That is, the branch condition is added to the path constraint.
3) l' is neither a start node nor a branch node. Then $\sigma' = (l', s \wedge c, \Pi, T')$, i.e., the constraint $c = [a(l)]$ is added for the symbolic variables. If l was an exit node, then the evaluation $c = [a(l)]$ continues with the return value for the respective function call expression.

IV. Architecture and Implementation

A. Main classes and Eclipse integration

The tool is a plug-in for the Eclipse IDE and extends the CDT code analysis framework (Codan [52]). Eclipse CDT provides a C/C++ parser and AST visitor, and the code analysis framework provides a control flow graph builder. Codan uses Eclipse's marker framework for reporting bugs in the CDT GUI.

The main classes of the architecture are illustrated in Figure 1. `WorkPoolManager` and `Worker` are active classes. The main functions of the classes are [12]:

- `WorkPoolManager` implements Codan's `IChecker` interface and through this becomes callable from the Eclipse GUI (through Codan). It starts `Worker` threads and reports found bugs through the Codan interface to the Eclipse marker framework. As synchronization object, `WorkPool` is used (synchronized methods). It tracks the number of active `Worker`s and serves for dynamic work re-distribution.
- `Worker` explores a part of the program execution tree specified by a start path, with help of its `Interpreter`. Different `Worker`-threads concurrently analyse disjunct partitions of a program's execution tree. `Worker` has a forward and a backward (backtracking) mode. It passes references to control flow graph nodes for entry (forward mode) or backtracking to its `Interpreter`.
- `Interpreter` performs symbolic interpretation according to the tree-based interpretation pattern [55]. The control flow node processor (`CFNodeProcessor`) implements CDT's AST visitor and translates into SMTlib logic equations.
- `SMTSolver` wraps the SMT solver. In the current implementation, the wrapper is configured to call the SMT solver described in [6].
- `IPathObserver` is an interface provided by the `Interpreter` for checker classes. Checkers can register for notifications and can access context information.
- `BranchValidator` checks branches for satisfiability with a solver query and throws an exception (caught by the `Worker`) in case of unsatisfiability, which causes pruning of the respective path. `BranchValidator` is triggered when entering a branch node.
- `ProgramStructureFacade` provides access to control flow graphs.

B. Tree-based interpretation

First, all source files of the program under test are parsed into ASTs, and a CFG is generated for each AST subtree

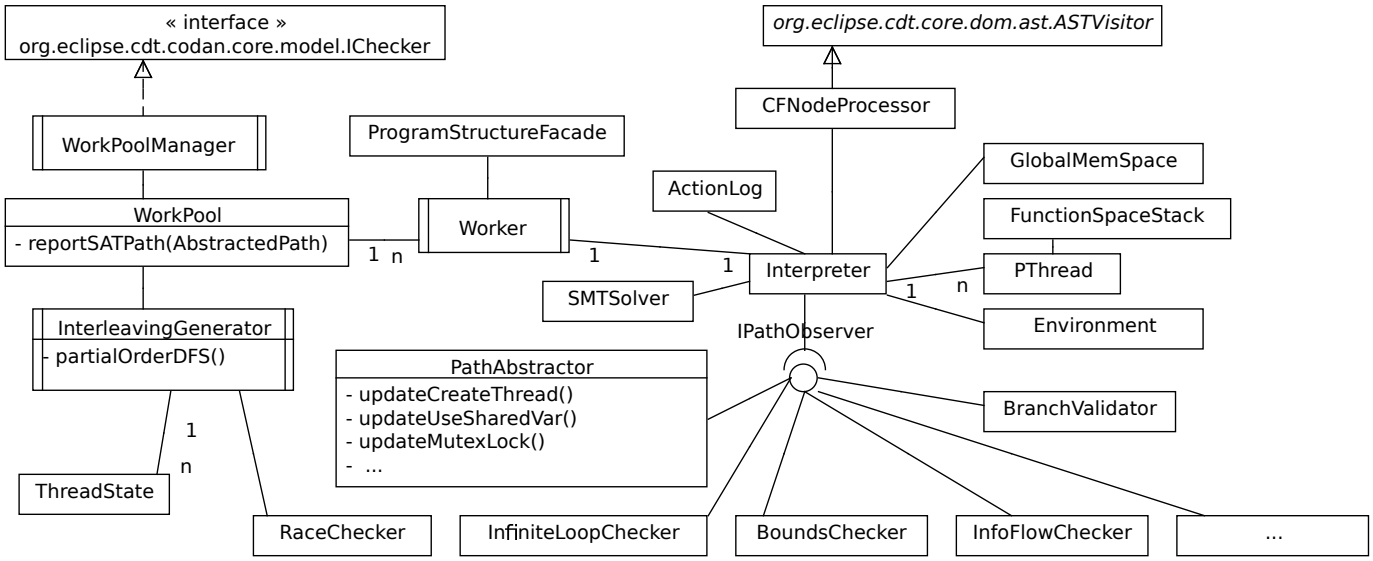


Figure 1. Architecture, main classes

that is rooted in a function definition. Symbolic execution traverses CFGs, beginning from `main()` start. For each control flow node, the corresponding AST subtree is interpreted. The maximum loop depth to be explored can be bounded. Symbolic variables are stored in and retrieved from a memory system, which consists of the classes `GlobalMemSpace` and `FunctionSpaceStack`. Symbolic variables are resolved by their syntax tree name (CDT's `IASTName`) and binding. Each `Interpreter` instance supports multiple thread objects (`PThread`) of which each has its own stack object (`FunctionSpaceStack`, compare Figure 1). Variable dependencies are traced. This is used for slicing, so that solver queries only contain the necessary subset of logic equations. The program under test communicates with its environment through the operating system API, which is wrapped by the C standard library. Symbolic execution is not extended into the standard library. Instead, symbolic function models can be provided for standard library functions (through the `Environment` class). In order to allow for backtracking of the symbolic program state, the semantic actions performed per CFG node, like, e.g., variable declarations, are stored in an `ActionLog`. The interpreter passes AST subtree references, which are referenced by CFG nodes, to `CFNodeProcessor` for translation.

Figure 2 illustrates the data structures for CFG and AST, which are provided by CDT/Codan. The figure shows on the left the control flow for an example function from [14]. On the right it shows two AST subtrees which are referenced by two control flow nodes. There are eight satisfiable paths through the function. One of these paths contains a buffer overflow bug. This path is depicted in red. All eight paths could be merged at the function exit. The function's exit node is depicted in blue.

In general, the interpretation works per CFG node. The current CFG node is interpreted, then a successor node is chosen, who is interpreted next. An exception are function calls. If a CFG node corresponds to a statement or expression that contains a function call, then the node is first only partially interpreted, i.e., up to the function call expression (which includes parameter collection). Then the called function's start node and respective successors are interpreted. After exit of the called function, interpretation continues with the remaining uninterpreted AST subtree part of the calling CFG node, with the function's return value.

C. Translation into SMTlib logic

Translation is implemented by bottom-up traversal of an AST subtree according to the visitor pattern [56]. This pattern is commonly used for operations on a graph of elements (here the AST), where the operation on a node depends on the node's runtime type. The class `CFNodeProcessor` implements CDT's `ASTVisitor` (compare Figure 1). Translation attributes are passed upwards during AST traversal. Attributes can be for example intermediate translation results. During translation, type promotion is performed according to the operators. The translation uses single assignments to avoid desctructive updates. Pointers and structs are not directly translated into SMT logic, they are represented internally during interpretation (e.g., a pointer has a target and an offset formula). Logic equations are generated at pointer dereference and at field access to a struct. The translation output are equation in the SMTlib sublogic of arrays, uninterpreted functions and nonlinear integer and real arithmetic (AUFNIRA). While the translation in general works per CFG node, one exception are function call expressions as mentioned in the last subsection. Another exception are `switch` statements – where the default branch's formula depends on all sibling CFG nodes.

D. Analysis of multi-threaded code

Analysis of multi-threaded code is supported for programs which use a subset of the Portable Operating System Interface (POSIX) threads API. Certain functions from the POSIX threads library like mutex locking and unlocking, and creation and joining of other threads are currently supported by function models [18]. The symbolic execution is run with a pre-defined

scheduling algorithm. The implementation uses 'lowest thread-id first' scheduling. A thread blocks (becomes inactive), e.g., by trying to acquire a lock already held by another thread or with a join call for a thread that is still alive. Unless the code contains race condition bugs, the program behaviour is identical for all possible schedulings, so one scheduling algorithm suffices.

If race conditions are also to be detected, a recording and abstraction of satisfiable paths can be activated. Scheduling can in principle occur between any assembler instructions. But most actions of different threads are independent and commutative. This can be exploited by a partial-order reduction [57]. Only thread interactions are relevant for race detection, where thread interactions are events like thread creation, joining, mutex locking and unlocking, and also the read or write access to shared variables. Whether or not a variable is shared between threads is in principle context-sensitive. It depends on the current program path including the current function's call context. All global variables are marked as shared when they are first accessed. Then the shared property is inferred over data flow constructs like assignments, references, function call parameters and return values. Further, the thread start arguments are also marked as shared. Shared variables are traced by the `Interpreter`, and all thread interaction events are recorded by the class `PathAbstractor`.

The analysis of multi-threaded code does not require more effort than the analysis of single-threaded code. Extra effort is only spent if race conditions are to be detected. Race condition analysis works on the recorded thread interaction abstraction level [18] and is described in more detail in Section V-D.

E. Multi-threaded engine

The implementation is multi-threaded, a configurable number of worker threads concurrently explores different parts of the execution tree [12]. Each worker performs a depth-first exploration of its partition with backtracking of the symbolic program state. Control flow graphs and syntax trees are shared between worker threads. AST nodes are not thread-safe. Workers therefore lock AST subtrees at the CFG node level, i.e., the AST subtree that is referenced by the currently interpreted CFG node. Dynamic redistribution of work between workers is enabled by splitting a workers 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 concatenations of the partition start path and one of the branches not taken by the current worker are returned as start paths for other workers. After a split, the current worker's start path is also prolonged by one branch node, which is the branch node that the worker had taken. The current worker's prolonged start path still defines its execution tree partition, which is now reduced in size. Analysis starts with one worker, who splits its partition until the configured number of workers is busy. A worker is initialized by replaying its partition start path. 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. The `WorkPool` is used for synchronization. It serves to exchange split paths between workers and tracks the number of active workers. A parallelization speed-up is possible if the program under test has decisions for which more than one branch is satisfiable [12].

F. Backtracking and path merging

1) Dead and live variables: Paths can be merged without any loss in bug detection accuracy when the path constraints and symbolic variable constraints differ only in dead variables [13]. The detection of such merge possibilities requires a context cache at potential merge points. Also required is a way to detect dead variables and to filter them from the path and variable constraints. Therefore, potentially interesting merge points are program locations where the sets of dead and live variables change. Such points are function start and function exit and after scope blocks like `if` / `else` or `switch` statements and loops.

2) Merge points: Path merges are performed at function exit in the current implementation. Merges are possible because stack frame variables die at function exit. Path and variable constraints at function exit are treated as concatenation of the function's call context and the local context. The approach misses possibilities to merge paths earlier after scope blocks inside one function. On the other hand, it does not require more complex live variable analysis at intermediate points. The approach merges paths that have split inside the same function, possibly with other function calls in between. It needs to know the set of variables that have been written since the merge paths have split. This is overapproximated by the set of variables written since entering the function that is left at the program location in question. A set of potentially read variables along path extensions is not computed. From the set of variables that have been written as local context (i.e., since function entry), global variables, the return value and all variables that have been written through pointers (pointer escape, potential write to, e.g., other stack frame) are assumed as live. The remaining written local variables are assumed as dead, which is a sound assumption. The local context is then reduced by removing the dead variables. A context cache is used to lookup observed reduced local contexts from pairs of a function's exit node (in the function's control flow graph) and call context. During symbolic execution, at each exit node the context cache is queried for a merge possibility. Then the current path is merged if possible, otherwise the local reduced context is added as new entry to the context cache.

3) Backtracking: Due to single assignment form, a symbolic program state contains all previous states along the path. Backtracking is enabled by class `ActionLog`, which records certain semantic actions performed for CFG nodes on the current path (e.g., variable creation or hiding). For example, if a function exit is backtracked, the function's stack frame with contained variables must be made visible again. Dead variables are therefore not garbage-collected, because this would impede backtracking. The engine further allows to record and visualize explored parts of a program execution tree.

4) Path merging: Path merging needs knowledge about the sets of written variables since path split. The implementation uses the class `ActionLog` to derive this information. It contains all writes to variables, including writes to globals and writes through pointers (potentially to other stack frames). The action log is looked through backwards up to the current

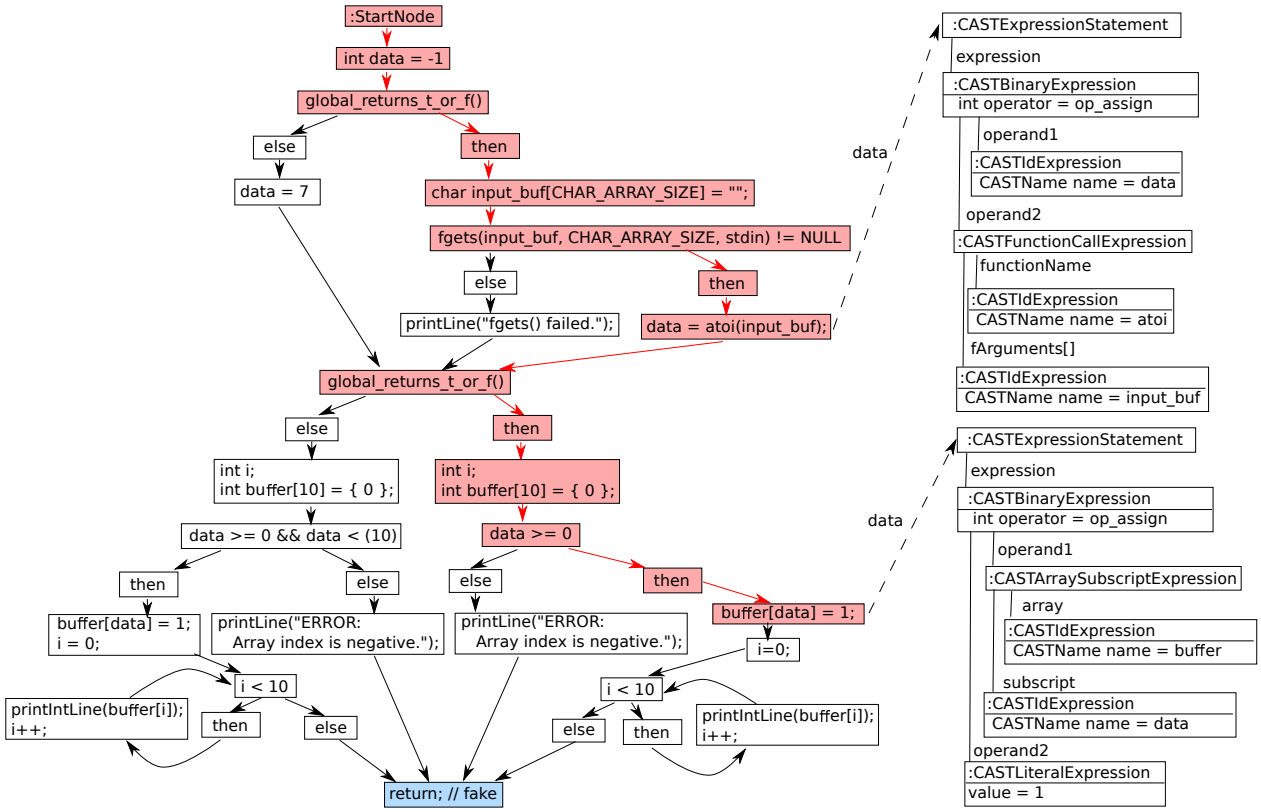


Figure 2. Left: Control flow graph for an example function from [14]. One path leading to a buffer overflow bug is marked red. Paths are merged at function exit (the function's exit node is marked blue). Right: Two AST subtrees referenced by CFG nodes

function's CFG start node, and the reduced local context is built from the variable declaration actions. The reduced local context is yielded by removing all writes to variables if the variables do not have global scope, are not written through pointers and are not the current function's return value. This approach does not necessitate a comparably more complex dead/live variable analysis. Path merge possibilities are detected using a class `ContextCache`, which is a `HashSet`. The keys are exit nodes with function call context, the values are the observed reduced local contexts. The context cache is queried at each function exit (CFG exit node). Comparing the reduced local contexts does not necessitate expensive calls to the SMT solver.

Path merging applies in the same way to branches that belong to loops, when the loop iteration number depends on program input (otherwise there would be only one satisfiable sub-path through the loop). Symbolic execution is currently applied with loop unrolling up to a maximum loop depth bound. A path through a loop can therefore split into a maximum number of paths equal to the loop unrolling bound. Branch nodes in the CFG belonging to loop statements are treated by symbolic execution just as branch nodes belonging to `if`/`else` statements. The branch nodes also have the same labels, i.e., '`then`' for the loop body and '`else`' to skip the loop. The only difference is that loops have a connector node with two incoming branches, which closes the loop before the decision node. However, this has no influence on the merging of unrolled paths.

V. Example Bug Checkers

A. Memory access

The major memory access bugs are stack-based or heap-based buffer over-write (overflow), over-read, under-write or under-read (CWE-121,122,124,126,127). The class `BoundsChecker` is triggered when the translation encounters array subscript expressions and pointer dereferences [12]. The bounds checker queries the set of equations, on which the pointer and offset variables depend, from the interpreter. Two satisfiability checks are then added to this equation system slice. One of them checks whether the offset could be negative (lower bound violation), the other checks whether the offset could be larger than the array size. These satisfiability queries are decided by the SMT solver.

B. Infinite loops

If an infinite loop can be triggered by unexpected program input, which is not properly validated, it can be used by an attacker for a denial of service attack. Since the standard number formats are discrete and finite, any infinite loop orbit without number overflow must be periodic. The common weakness enumeration calls this 'loop with unreachable exit condition' (CWE-835), in contrast to 'number overflow' bugs. An infinite loop can therefore be detected with a fixed-point satisfiability check. It checks whether it is satisfiable that the loop is re-visited with identical context. The class `InfiniteLoopChecker` is trigger for 'loop closed' events, i.e., when a decision node is re-visited on the path [16]. The

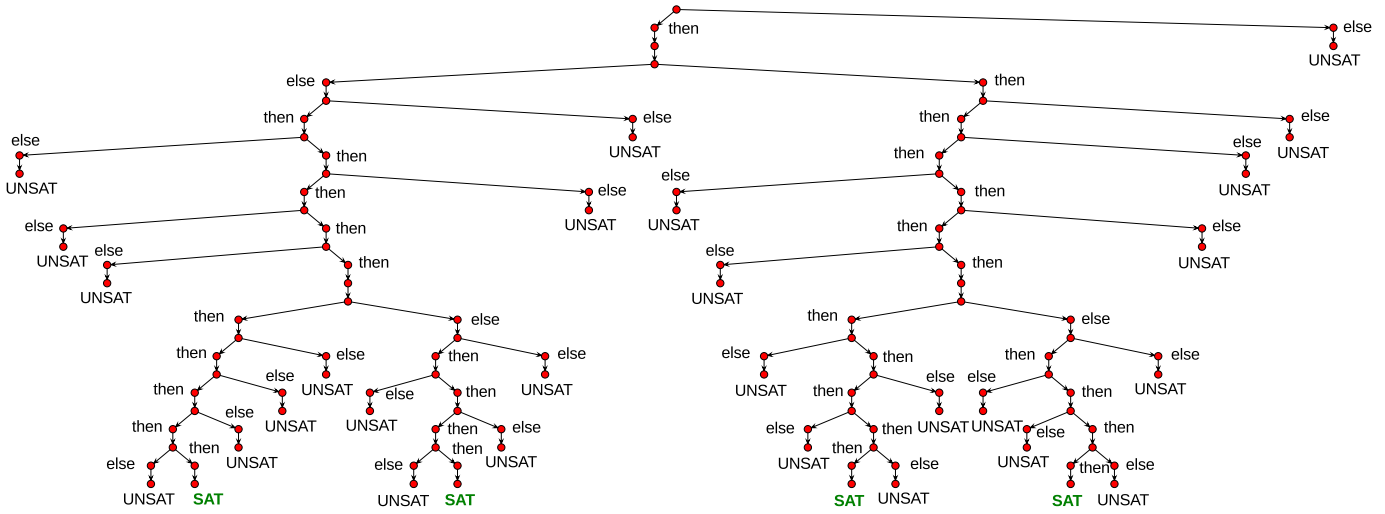


Figure 3. Execution tree for test program `CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy_12` from [14], showing only decision and branch nodes.

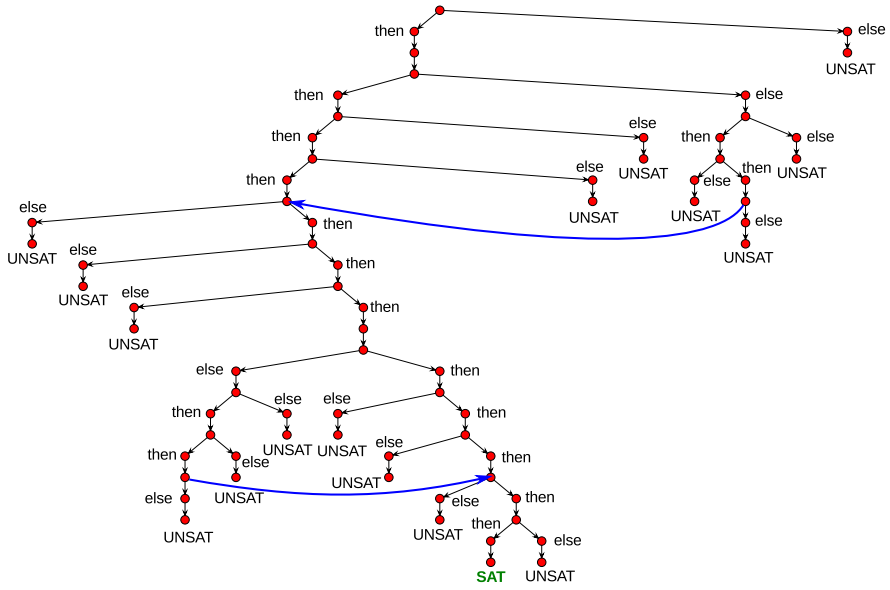


Figure 4. Effect of path merging for the test program of Figure 3. The execution tree is folded at two locations (blue arrows). The number of traversed satisfiable paths is reduced from four to one.

loop variables are identified using the `ActionLog`. The loop checker formulates the fixed-point query, which is then passed to the solver.

The loop checker avoids re-checking 'simple' loops in different contexts by performing context-free termination and non-termination checks at the loop's first closing event. This check is performed without the constraint of the path on which the loop is reached, i.e., only using the loop guard set and the unrolled loop body (unrolled one iteration). The termination check for 'simple' loops is based on Brouwer's fixed-point theorem [16], [58]. This theorem implies that all linear loops that do not have a fixed-point in the guard set must terminate.

Loops that have not been decided by the context-free checks are checked context-sensitively during symbolic execution, i.e., with consideration of the path and variable constraints of the path on which the loop is reached. Symbolic execution unrolls all satisfiable paths through the loop, up to a configurable loop depth bound. Like for other checkers with symbolic execution, bug detection is sound (no false positives) and bounded complete [16]. The infinite loop checker detects all infinite loops with t prefix loop iterations and a loop orbit periodicity of $p \leq n - t$, when all loops are unrolled up to a depth n [16].

C. Information flow

Examples for information flow bugs are cleartext transmission of sensitive information (CWE-319) or information exposures through environment variables, debug log files or shell error messages (CWE-526, 534, 535 [2]). The model for secure information flow follows the lattice model from [59]. Information belongs to 'security classes', and information must not flow from higher to lower security classes. The implemen-

tation regards the operating system API as trust boundary [17]. Information flows through a program from sources to sinks, which are standard library calls and therefore trust boundaries. These trust boundaries are implemented in the function models, which are accessed through the `Environment` classs (compare Figure 1). Program input is labelled with a security class. During symbolic interpretation, security class affiliation is inferred over the data flow. The `InfoFlowChecker` is triggered when a trust boundary is crossed with program output, i.e., for information sinks. The checker assures that sensitive output parameters do not flow into a lower security sink [17].

D. Atomicity violations

The detection is based on thread interleavings, that is, possible thread schedulings on a single-core processor. The detection is implemented on the abstraction level of thread interactions, from the recorded satisfiable program paths with 'lowest thread-id first' scheduling [18]. Alternative thread interleavings are generated by class `InterleavingGenerator`, and the detection of atomicity violations (CWE-366) is implemented in class `RaceChecker` (compare Figure 1). The `InterleavingGenerator` generates the scheduling tree of relevant alternative thread interleavings from the abstracted satisfiable program paths. The algorithm uses ample set partial order reduction [60] and selectable interleaving coverage [61].

1) Ample set partial order reduction: From a satisfiable program path, all other thread interleavings corresponding to different scheduling decisions can be generated. The generated set of interleavings should be of minimal size without degrading the ability to detect atomicity violations. The tree of possible scheduling decisions is traversed on-the-fly with depth-first search. The tree nodes are constructed as maximal sets of independant actions (ample sets). The construction of ample sets reduces the width of the scheduling tree and thus the number of generated interleavings. `read()` or `write()` actions from different threads for shared variables are independent if the variable is not the same. Actions may be blocked until they are enabled by other actions. Examples are a thread waiting to acquire a lock held by another thread, or a thread waiting to join another. Interleaving representatives are found as tree leafs, i.e., when there are no more blocked and enabled actions. The representative is given as path through the ample set scheduling tree [18].

2) Interleaving coverage: Like there are different code coverage criteria for single-threaded code, e.g., branch coverage or modified condition/decision coverage, there are also different interleaving coverage criteria for multi-threaded code [61]. In general, concurrency bugs can involve any number of threads and variables. However, due to its practical relevance, the special case of atomicity violations as overlapping `read()`/`write()` actions to the same variable from different threads is of particular interest. Therefore, the interleaving generation not only supports the 'all interleavings' criterion (with partial order reduced implementation), but currently also the 'local-or-remote-define' criterion from [61]. This criterion means that for every read access to a shared variable, both an interleaving where the respective variable was defined in the local thread and one interleaving where it was defined by a remote thread are covered. This criterion offers a far better scaling behaviour, at the expense of missing more involved concurrency bugs. Reduced interleaving coverage is implemented jointly with partial order reduction as 'branch and bound' pruning of the ample set scheduling tree.

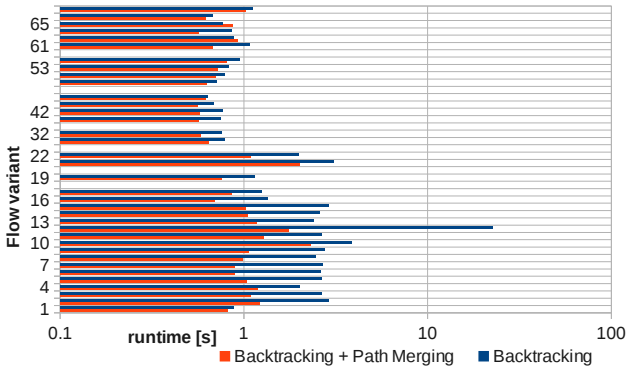
3) Atomicity violation detection: Atomicity violations are detected in the set of generated interleaving representatives by the class `RaceChecker`, which looks for overlapping `read()`/`write()` actions (at least one read and two writes) from different threads to the same variable [18].

VI. Experiments

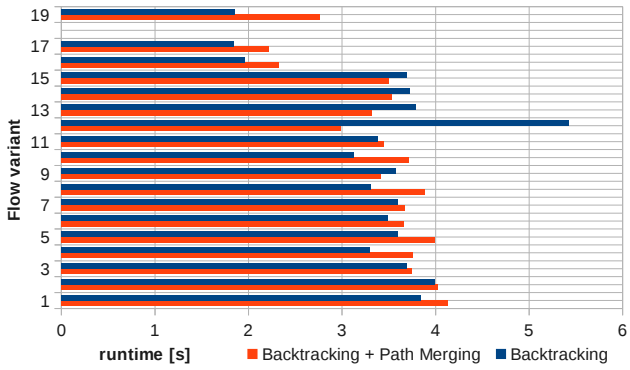
The tool is evaluated with test cases from the Juliet suite for C/C++ [14]. The test programs are artificial and automatically generated by combination of baseline bugs and control/data flow variants, in order to cover all language constructs. In the current version (1.2), the suite covers 1617 baseline bugs (flaw types) for 118 common weaknesses. Combined with 48 flow variants (38 of them for C, 10 only for C++) this results in over 60000 test cases (buggy programs) with together over 8 million lines of code. Bugs are context sensitive. The maximum bug context depth needed for accurate detection, i.e., no false positive and no false negative detections, is 5 function calls in 5 different source files (flow variant 54). Flow variants include flow controlled by global variables, different loop types, function pointers or void pointers,

Analysis of test programs could be started manually through the Eclipse GUI, i.e., through the extensions provided by Codan [52]. Codan in turn calls the symbolic execution plug-in presented in this paper (if activated in the Codan configuration). A screenshot for bug reporting with the CDT GUI is shown in Figure 6. In order to measure analysis run-times, the tests are rather run as JUnit plug-in tests. The measurements are obtained with Eclipse 4.3 on 64bit Linux kernel 3.2.0 and an i7-4770 CPU. This section evaluates the effect of path merging on analysis run-times. The same bug detection accuracy with and without path merging is validated, there are no false positive or false negative bug detections on the test set.

The effect of path merging on the execution tree is illustrated with the test program `CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy_12`, which denotes a buffer overflow with `memcpy()` and flow variant 12 [14]. It contains a 'good' and a 'bad' function. The 'bad' function is shown in a slightly simplified version in listing 1. The function contains an `if`/`else` decision for which both branches are satisfiable. In the `then` branch it contains a buffer overflow bug, which is marked with a comment in the listing. For both branches the function only writes to stack variables, and the reduced local context at function exit is the empty set for both branches. Merging the two paths at function exit, which have split at the decision node, is therefore clearly possible without missing any bug. The 'good' function is almost identical, but is bug-free. Apart from some output functions, this program calls the 'good' and 'bad' function once each. Therefore, it contains four satisfiable paths. The execution tree is illustrated in Figure 3. The figure only shows decision nodes and branch nodes. Therefore, the top node in the figure is the first decision



(a) Analysis time for buffer overflow tests with `fgets()`.



(b) Analysis time for race condition tests on global variables.

Figure 5. Analysis run-times for test sets `CWE121_fgets` and `CWE366_global_int` from [14].

TABLE I. Analysis runtime sums for five test sets from [14], with and without path merging.

	CWE121_fgets (36 test programs)	**CWE121_memcpy** (18 test programs)	**CWE366_global_int** (18 test programs)	**CWE366_int_byref** (18 test programs)	**CWE835** (6 test programs)	**Sum** (96 test programs)
backtracking	80,7 s	14,7 s	61,2 s	62,1 s	9,0	227.7 s
backtracking + path merging	34,4 s	15,3 s	59.2 s	62.6 s	9,8	181.3 s

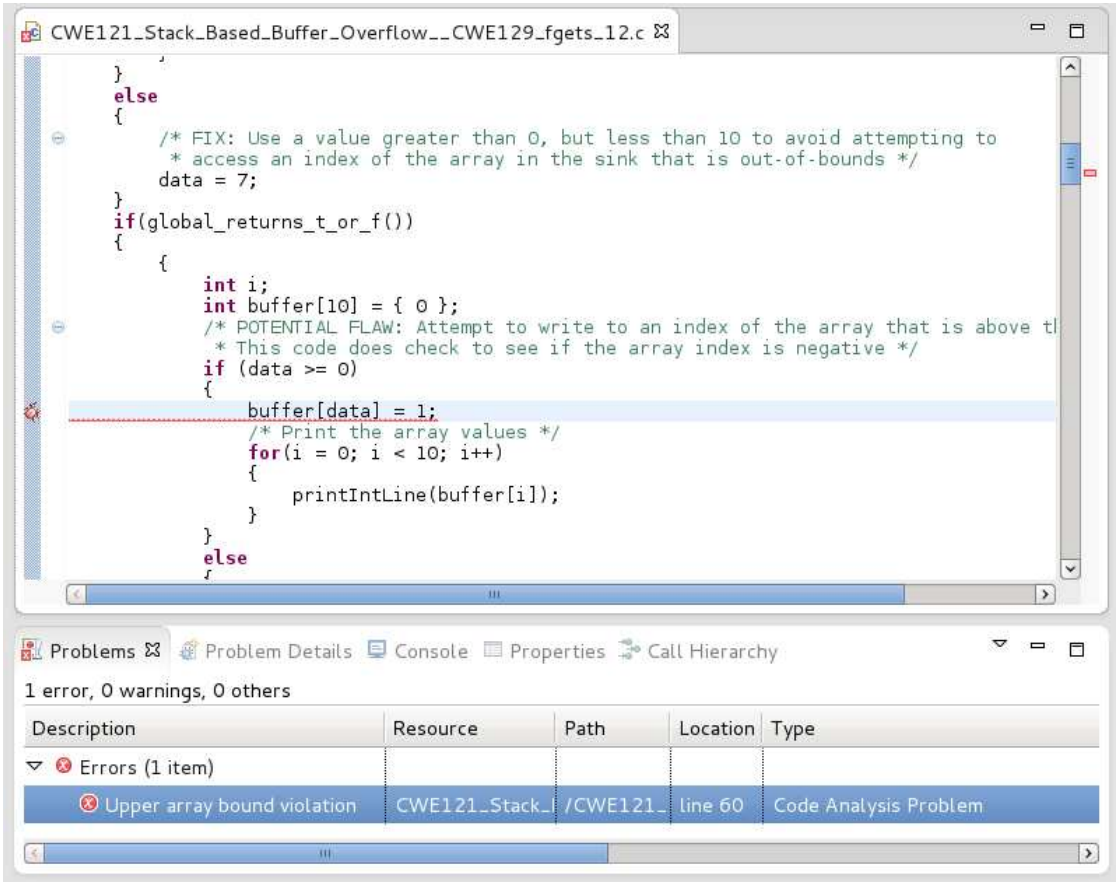


Figure 6. Bug reporting in the Eclipse GUI.

node after start of the `main()` function. Branch nodes are marked with the branch label (`if` or `else`). If a branch is decided to be unsatisfiable, the figure indicates this with a child note marked `UNSAT`. If the program end is reached, the figure indicates this with a child node marked `SAT`. Figure 4 shows the same tree when path merging is applied. Paths are merged at two points (the two function exits), which is indicated in the tree with blue arrows. An arrow connects two paths, which are merged. The arrow source indicates the subtree that is pruned, the arrowhead indicates the subtree that is further traversed. It can be seen that path merging corresponds to folding of the execution tree, the number of traversed satisfiable paths is reduced from four to one.

An infinite loop example from [14] is shown in Listing 2. The function contains a single-path loop, which is non-terminating for all input. While path merging could in principle be applied to infinite loops, a checker is still used to detect the infinite loop as software bug. The example loop is decided with the context-free non-termination test for 'simple loops' [16], unrolling the loop only once (rather than 256 times).

Listing 1. Simplified example function from [14], contains a buffer overflow in the `then` branch.

```
typedef struct _charvoid
{
    char x[16];
    void * y;
    void * z;
} charvoid;

void CWE121_memcpy_12_bad_simplified() {
  if(global_returns_t_or_f()) {
    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));
    /* null terminate the string */
    cv_struct.x[(sizeof(cv_struct.x)/sizeof(
       char))-1] = '\0';
  }
  else {
    charvoid cv_struct;
    cv_struct.y = (void *)SRC_STR;
    /* FIX: Use sizeof(cv_struct.x) to avoid
       overwriting the pointer y */
    memcpy(cv_struct.x, SRC_STR,
       sizeof(cv_struct.x));
    /* null terminate the string */
    cv_struct.x[(sizeof(cv_struct.x)/sizeof(
       char))-1] = '\0';
  }
}
```

Table I shows analysis benchmark results with and without path merging for five test sets (in sum 96 programs) from [14], which contain buffer overflows, races and infinite loops bugs. The effect of path merging varies per test-set. Path merging requires a certain overhead for computing and comparing reduced local contexts. If path merging possibilities are found, there is a speed-up of analysis time. The table shows a 60% speed-up for tests containing buffer overflows with `fgets()`

Listing 2. Example infinite loop from [14].

```
void CWE835_Infinite_Loop__do_01_bad() {
  int i = 0;
  do {       /* FLAW: no break */
    printIntLine(i);
    i = (i + 1) % 256;
  } while(i >= 0);
}
```

(from 80.7 s to 34.4 s), but a 9% slow-down for the infinite loop tests.

Figure 5 shows the analysis runtimes for the sets of buffer overflows with `fgets()` (Figure 5a) and for the races test set on global variables (Figure 5b). The figure shows the runtimes depending on the test case data/control flow variant, for the symbolic execution engine with backtracking only and for backtracking with path merging. Figure 5a uses a logarithmic scale and contains values for 36 flow variants. Flow variants in Juliet are not numbered consecutively, to leave room for later insertions. Since path merging folds complete subtrees of a program's execution tree, it has an exponential effect on runtimes. This is exemplified by flow variant 12. While merging paths for the `memcpy()` buffer overflow with variant 12 reduces the runtime only from 1.1 s to 0.8 s, the runtime for the `fgets()` buffer overflow is reduced from 22.8 s (longest analysis time for any tested program) to 1.7 s. This is because the `fgets` test program contains several other decision nodes with two satisfiable branches.

The dependence of possible path merging speed-up on the program structure becomes clear through the specific test case structure, which is a combination of baseline flaws with flow variants. There are three possibilites:

1) The baseline flaw has a path merging possibility. Then it is likely that there is a speed-up already for the simplest test program containing the bug (flow variant 1), and for all other flow variants. An example is the `CWE121_fgets` test set (compare Figure 5a).
2) The baseline flaw does not have a path merging possibility. Then there can only be a speed-up for test cases, in which the flow variant contains a merging possibility. In the current test suite version this is only the case for flow variant 12. An example is the `CWE366` test set, where only flow 12 shows a significant path merging speed-up (compare Figure 5b).
3) Neither the baseline flaw nor any flow variants compatible with this flaw contain merging possibilites. An example is the infinite loop test set (`CWE835`, compare Table I).

VII. Discussion

This paper describes a backtracking symbolic execution engine with path merging functionality and its implementation in Eclipse CDT. Symbolic execution enables sound bug detection, i.e., without false positives. The evaluation of path merging with small test programs from the Juliet suite already shows a significant speedup. For larger programs, path merging has an exponential effect on analysis runtimes (exponential in

the number of decision nodes with more than one satisfiable branch). Future work might include the investigation of the effect of additional merge points inside functions. Automated abstraction might enable scaling to larger programs by offering yet more path merging possibilities. The implementation could also be used as a basis for selective symbolic execution, e.g., by adding consistent concrete execution using CDT's debugger services framework. Another direction is the automated generation of quick-fix refactoring proposals based on the obtained information about bugs and program paths on which they occur. The tight tool integration enabled by Eclipse seems advantageous for this purpose.

ACKNOWLEDGEMENT

This work was funded by the German Ministry for Education and Research (BMBF) under grant 01IS13020.

REFERENCES

[1] A. Ibing, "A backtracking symbolic execution engine with sound path merging," in Int. Conf. Emerging Security Information, Systems and Technologies, 2014, pp. 180–185.

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

[3] J. King, "Symbolic execution and program testing," Communications of the ACM, vol. 19, no. 7, 1976, pp. 385–394.

[4] L. deMoura and N. Bjorner, "Satisfiability modulo theories: Introduction and applications," Communications of the ACM, vol. 54, no. 9, 2011, pp. 69–77.

[5] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard version 2.0," in Int. Workshop Satisfiability Modulo Theories, 2010.

[6] L. deMoura and N. Bjorner, "Z3: An efficient SMT solver," in Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 337–340.

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

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in Int. Symp. Code Generation and Optimization (CGO), 2004, p. 75.

[9] T. Kremenek, "Finding software bugs with the Clang static analyzer," LLVM Developers' Meeting, Aug. 2008, retrieved: 05/2015. [Online]. Available: http://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf

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

[11] 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), 2006, pp. 322–335.

[12] A. Ibing, "Parallel SMT-constrained symbolic execution for Eclipse CDT/Codan," in Int. Conf. Testing Software and Systems (ICTSS), 2013, pp. 196–206.

[13] 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), 2008, pp. 351–366.

[14] T. Boland and P. Black, "Juliet 1.1 C/C++ and Java test suite," IEEE Computer, vol. 45, no. 10, 2012, pp. 88–90.

[15] TIOBE index, retrieved: 05/2015. [Online]. Available: www.tiobe.com/index.php/content/paperinfo/tcpi

[16] A. Ibing, "A fixed-point algorithm for automated static detection of infinite loops," in IEEE Int. Symp. High Assurance Systems Eng., 2015, pp. 44–51.

[17] 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, pp. 84–93.

[18] A. Ibing, "Path-sensitive race detection with partial order reduced symbolic execution," in Workshop on Formal Methods in the Development of Software, 2014, pp. 311–322.

[19] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," Communications of the ACM, vol. 56, no. 2, 2013, pp. 82–90.

[20] C. Cadar et al., "Symbolic execution for software testing in practice – preliminary assessment," in Int. Conf. Software Eng., 2011, pp. 1066–1071.

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

[22] 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, vol. 86, no. 8, 2013, pp. 1978–2001.

[23] 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. [Online]. Available: http://frama-c.com/download/acsl.pdf

[24] 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, vol. 7, no. 3, 2005, pp. 212–232.

[25] 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. [Online]. Available: http://frama-c.com/download/frama-c-user-manual.pdf

[26] 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, 2009, pp. 23–42.

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

[28] 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, 2004, pp. 168–176.

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

[30] 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), 1977, pp. 238–252.

[31] K. Zurowska and J. Dingel, "Symbolic execution of UML-RT state machines," in ACM Symp. Applied Computing, 2012, pp. 1292–1299.

[32] H. Hansen, J. Ketema, B. Luttik, M. Mousavi, and J. Pol, "Towards model checking executable UML specifications in mCRL2," Innovations Syst. Softw. Eng., no. 6, 2010, pp. 83–90.

[33] J. Abrial and L. Mussat, "Introducing dynamic constraints in B," in B Conference, 1998, pp. 83–128.

[34] 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., 2010, pp. 445–454.

[35] N. Tillmann and J. Halleux, "Pex – white box test generation for .NET," in Int. Conf. Tests and Proofs (TAP), 2008, pp. 134–153.

[36] W. Visser, C. Pasareanu, and S. Khurshid, "Test input generation with Java PathFinder," in Int. Symp. Software Testing and Analysis (ISSTA), 2004, pp. 97–107.

[37] 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, 2008, pp. 1–25.

[38] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz, "BAP: A binary

analysis platform," in Int. Conf. Computer Aided Verification, 2011, pp. 463–469.

[39] P. Godefroid, M. Levin, and D. Molnar, "Automated whitebox fuzz testing," in Network and Distributed System Security Symp. (NDSS), 2008.

[40] 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, 2005, pp. 263–272.

[41] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in Conference on Programming Language Design and Implementation, 2005, pp. 213–223.

[42] 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, 2011.

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

[44] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in Int. Conf. Computer Aided Verification (CAV), 1997, pp. 72–83.

[45] E. Clarke, O. Grumberg, Y. Lu, S. Jha, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," Journal of the ACM, vol. 50, no. 5, 2003, pp. 752–794.

[46] J. Jaffar, A. Santosa, and R. Voicu, "An interpolation method for CLP traversal," in Int. Conf. Principles and Practice of Constraint Programming (CP), 2009, pp. 454–469.

[47] K. McMillan, "Lazy annotation for program testing and verification," in Int. Conf. Computer Aided Verification (CAV), 2010, pp. 104–118.

[48] J. Jaffar, J. Navas, and A. Santosa, "Unbounded symbolic execution for program verification," in Int. Conf. Runtime Verification, 2011, pp. 396–411.

[49] J. Jaffar, V. Murali, J. Navas, and A. Santosa, "TRACER: A symbolic execution tool for verification," in Int. Conf. Computer Aided Verification (CAV), 2012, pp. 758–766.

[50] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," The Journal of Symbolic Logic, vol. 22, no. 3, 1957, pp. 269–285.

[51] OSGi Alliance Specifications, retrieved: 05/2015. [Online]. Available: www.osgi.org/Specifications

[52] A. Laskavaia, "Codan- C/C++ static analysis framework for CDT," in EclipseCon, 2011.

[53] D. LeBerre and A. Parrain, "The SAT4J library, release 2.2, system description," Journal on Satisfiability, Boolean Modeling and Computation, no. 7, 2010, pp. 59–64.

[54] R. Dannenberg and G. Ernst, "Formal program verification using symbolic execution," IEEE Trans. Software Eng., vol. 8, no. 1, 1982, pp. 43–52.

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

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

[57] E. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," Int. J. Software Tools for Technology Transfer, vol. 2, no. 3, 1999, pp. 279–287.

[58] L. Brouwer, "Über Abbildungen von Mannigfaltigkeiten," Mathematische Annalen, no. 71, 1911.

[59] D. Denning, "A lattice model of secure information flow," Communications of the ACM, vol. 19, no. 5, 1976, pp. 236–243.

[60] D. Peled, "Combining partial order reduction with on-the-fly model-checking," in Workshop on Computer Aided Verification, 1994.

[61] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," in ECEC/FSE, 2007.