A Backtracking Symbolic Execution Engine with Sound Path Merging

Andreas Ibing

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

Abstract—**Software vulnerabilities are a major security threat and can often be exploited by an attacker to intrude into systems. 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. If applied with approximate path coverage, it faces the state explosion problem. The number of paths in the program execution tree grows exponentially with the number of decision nodes in the program for which both branches are satisfiable. In combination with the standard approach using the worklist algorithm with state cloning, this also leads to exponential memory consumption during analysis. This paper considers a source-level symbolic execution engine which uses backtracking of symbolic states instead of state cloning, and extends it with a sound method for merging redundant program paths, based on live variable analysis. An implementation as plug-in extension of the Eclipse C/C++ development tools (CDT) is described. The resulting analysis speedup through path merging is evaluated on the buffer overflow test cases from the Juliet test suite for static analyzers on which the original engine had been evaluated.**

Keywords–Static analysis; Symbolic execution.

I. Introduction

Software vulnerabilities like, e.g., buffer overflows can in many cases be exploited by an attacker for remote code execution. Automated bug detection during software development and for releases are a main component of application security assurance.

Symbolic execution [1] is a static 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 then used to check program paths for satisfiability and to check error conditions for satisfiability. The current state of automatic theorem provers are Satisfiability Modulo Theories (SMT) solvers [2], the standard interface is the SMTlib [3]. An example state-of-the art solver is [4].

Automatic analysis tools which 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. Examples are [5] and [6], which analyzes LLVM code [7]. An overview of available tools is given in [8][9][10]. 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 [11] or to generate quick-fix proposals. An example of a source-level symbolic execution engine for C/C++ is [12], which uses the parser and control flow graph (CFG) builder from Eclipse CDT [13].

During symbolic execution, the engine builds and analyzes satisfiable paths through programs, where paths are lists of 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 [14]. Symbolic program states of frontier nodes (unexplored nodes) of the program execution tree are kept in memory, and at program branches the respective states are cloned. The reuse of intermediate analysis results with state cloning has the downside of being memory-intensive. [5] uses state cloning with a recursive data structure to store only state differences. 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 program execution tree grows exponentially with the number of decisions in the program for which both branches are satisfiable. Straight-forward application of symbolic execution with approximate path coverage (where the number of unrolled loop iterations is bounded) is therefore not scalable. This is often called the path explosion problem. In [15] 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 which extends [5]. 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 negative detections) and completeness (no false positives). 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

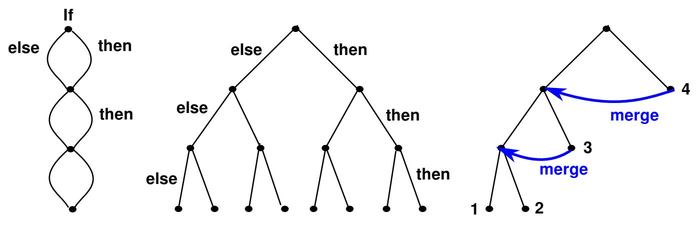


Figure 1. Sequence of three decisions and corresponding branches (left); the execution tree under the assumption that all branches are satisfiable splits into $2^3 = 8$ leafs (middle); path merging folds the execution tree (right).

C/C++ bug test suite for static analyzers is the Juliet suite [16]. Among other common software weaknesses [17] it contains buffer overflow test cases. 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 languages grammar constructs and to test the context depth of the analysis. The maximum context depth spanned by a flow variant is five functions in five different source files.

This paper develops and evaluates a sound path merging method in a source-level backtracking symbolic execution engine. The implementation extends [12]. The remainder of this paper is organized as follows. Section II describes the design decisions. Section III gives an overview of the implementation in Eclipse CDT. Section IV presents results of experiments with buffer overflow test cases from the Juliet suite. Section V discusses related work and section VI then discusses the presented approach based on the results.

II. Merge Points and Context Cache

A. Dead and live variables

Paths can be merged without any loss in bug detection accuracy when the path constraints differ only in dead variables. 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 constraint. Potentially interesting merge points are therefore 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.

B. Design decisions

The idea of merging program paths during symbolic execution is illustrated in Figure 1. The left of the figure shows a control flow with a sequence of three decisions and corresponding branches. For the assumption that all branches are satisfiable, the middle of the figure shows the execution tree which splits into $2^3 = 8$ leafs. The right of the figure illustrates how path merging potentially folds the execution tree together again. In this work, path merges are performed at function exit. Merges are possible because stack frame variables die at function exit. A path constraint at function exit is 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

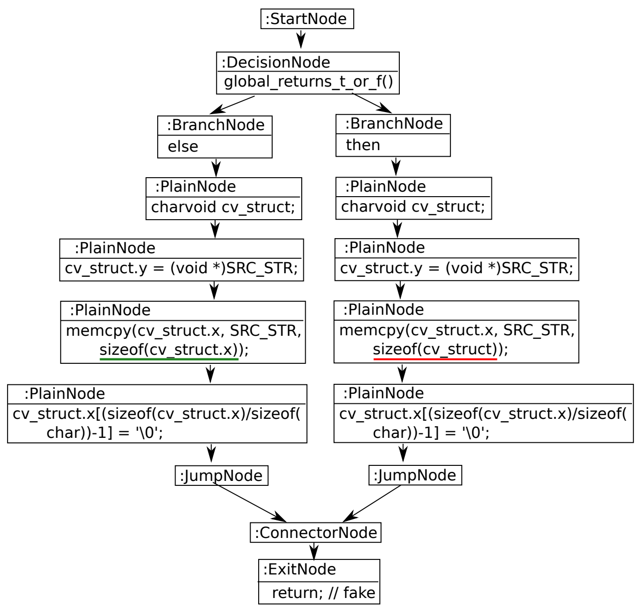

:StartNode
:DecisionNode
global_returns_t_or_f()
:BranchNode
else
:BranchNode
then
:PlainNode
charvoid cv_struct;
:PlainNode
charvoid cv_struct;
:PlainNode
cv_struct.y = (void *)SRC_STR;
:PlainNode
cv_struct.y = (void *)SRC_STR;
:PlainNode
memcpy(cv_struct.x, SRC_STR, sizeof(cv_struct.x));
:PlainNode
memcpy(cv_struct.x, SRC_STR, sizeof(cv_struct));
:PlainNode
cv_struct.x[(sizeof(cv_struct.x)/sizeof(char))-1] = '\0';
:PlainNode
cv_struct.x[(sizeof(cv_struct.x)/sizeof(char))-1] = '\0';
:JumpNode
:JumpNode
:ConnectorNode
:ExitNode
return; // fake

Figure 2. Control flow graph for example function from Figure 3. with buffer overflow in the `then` branch.

at intermediate points. The approach merges paths which have split inside the same function, possibly with other function calls in between. It needs to know the set of variables which have been written since the merge paths have split. This is overapproximated by the set of variables written since entering the function which 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 which have been written as local context (i.e., since function entry), global variables, the return value and all variables which have been written through pointers (pointer escape, potential write to other stack frame etc.) are assumed as live. The remaining written local variables are soundly assumed as dead. 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 pruned (merged) if possible, otherwise the local reduced context is added as new entry to the context cache.

III. Implementation in Eclipse CDT

A. Symbolic execution with symbolic state backtracking

This subsection shortly reviews [12] which is extended by the paper at hand with path merging functionality. The backtracking symbolic execution engine [12] uses Eclipse CDT's C/C++ parser to construct abstract syntax trees (AST) from source code files. Control flow graphs (CFG) are then constructed for function definitions rooted in AST subtrees. CFG construction uses the `ControlFlowGraphBuilder` class from CDT's code analysis framework (Codan [13]). The symbolic interpretation is implemented according to the tree-based interpretation pattern from [18], the translator class extends CDT's `ASTVisitor` (visitor pattern [19]). The interpretation is symbolic, i.e., variable values are logic formulas. Satisfiability queries to the SMT solver use the SMTLIB

```
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';
  }
}
```

Figure 3. Simplified example function from [16], contains a buffer overflow in the `then` branch. Corresponding CFG in Figure 2.

sublogic of arrays, uninterpreted functions and nonlinear integer and real arithmetic (AUFNIRA). Backtracking is enabled by a class `ActionLog` which records certain semantic actions performed for CFG nodes on the current path (e.g., variable creation or hiding). If for example a function exit is backtracked, the function's stack frame with contained variables must be made visible again. Dead variables are therefore not garbage-colled, because this would impede backtracking. The engine further allows to record and visualize explored parts of a program execution tree. The engine was evaluated in [12] by measuring detection accuracy (false positives and false negatives) and run-times for the detection of buffer overflows in Juliet test programs.

B. Path merging

In this implementation, paths are merged at function exit. The method can merge paths which have split since entering the same function, with the possibility that several other functions are called between entering and leaving the function. Path merging needs knowledge about the sets of written variables since path split. The implementation uses the class `ActionLog` from [12] 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 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 don't 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.

An example function is shown as listing in Figure 3. It is a simplified version of a 'bad' function from one of the buffer overflow test cases of the Juliet suite. The control flow graph of this function is shown in Figure 2. The function contains a decision node corresponding to an `if`/`else` statement, for which both branches are satisfiable. The error location is marked by red underlining in the branch on the right of Figure 2. and by 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. Merging the two paths at function exit which have split at the decision node is therefore clearly possible without missing any bug.

Path merging applies in the same way to branches which 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, `'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. This however has no influence on the merging of unrolled paths.

IV. Experiments

Path merging is evaluated on the same buffer overflow test programs from the Juliet suite as [12]. These programs contain buffer overflows with the `memcpy` (18 programs) and `fgets` (36 programs) standard library functions, and cover the Juliet control and data flow variants for C (e.g., multipath loops and fuction pointers). A screenshot for error reporting with the CDT GUI is shown in Figure 7. The tests are run as JUnit plug-in tests with Eclipse 4.3 on 64bit Linux kernel 3.2.0 and an i7-4770 CPU. 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.

Figures 4. and 5. illustrate the merging of paths, which corresponds to folding the execution tree. Figure 4. shows the execution tree for a `memcpy` buffer overflow with flow variant 12. This test program contains a 'good' and a 'bad' function, where both functions contain a decision node with two satisfiable branches. The bad function is given in a simplified version in Figure 3. The tree shows only decision nodes and branch nodes. Figure 5. shows the same tree when path merging is applied. Paths are merged at two points which are indicated in the tree (the two function exits), and the traversal of two subtrees is skipped.

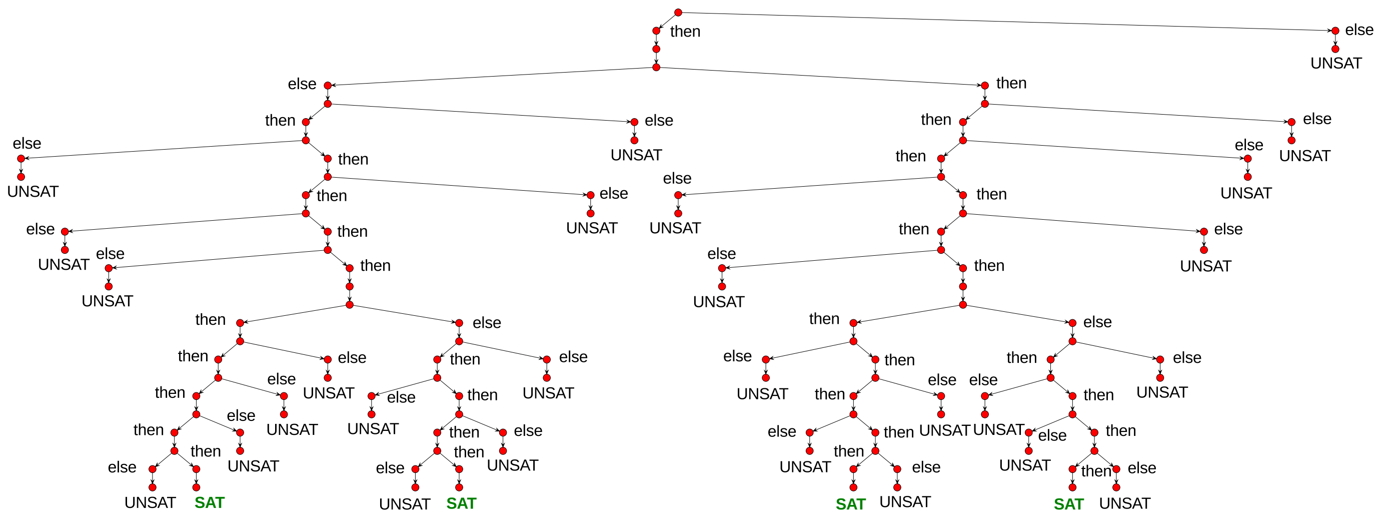


Figure 4. Execution tree for test program `CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy_12` from [16], showing only decision and branch nodes.

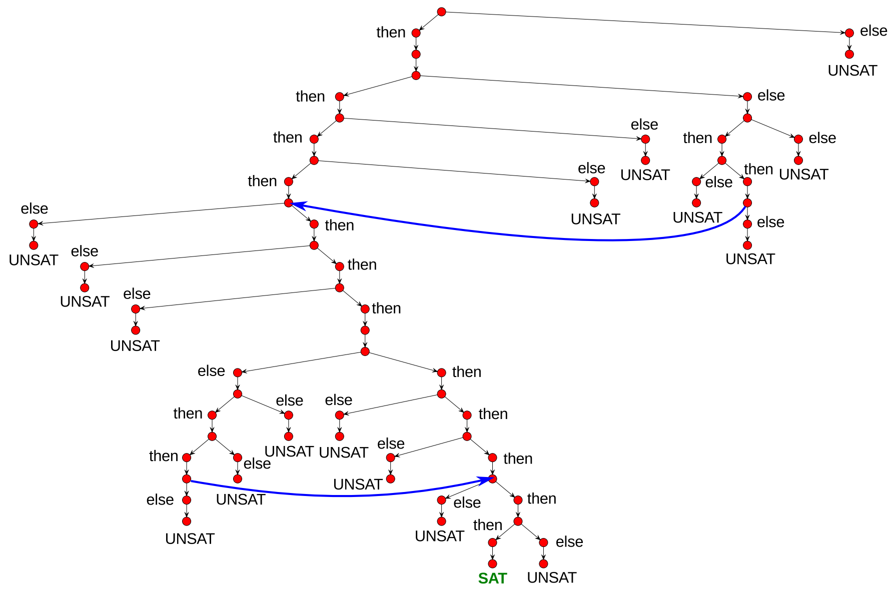


Figure 5. Effect of path merging for the test program of Figure 4. The execution tree is folded at two locations. The number of traversed satisfiable paths is reduced from four to one.

TABLE I. Analysis runtime sums for the two test sets, with and without path merging.

	CWE121_memcpy (18 test programs)	**CWE121_CWE129_fgets** (36 test programs)	**Sum** (54 test programs)
backtracking according to [12]	14,7 s	80,7 s	95,4 s
backtracking and path merging	15,3 s	34,4 s	49,7 s

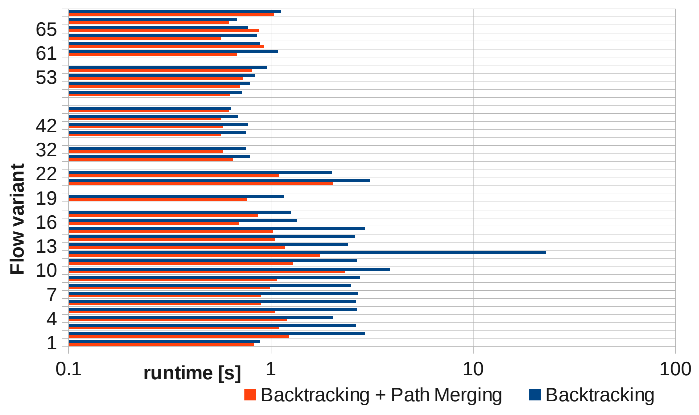


Figure 6. Analysis runtimes with and without path merging for 54 buffer overflow test programs from [16], with corresponding control/data flow variant numbers.

Figure 6. shows the analysis runtimes for the set of buffer overflows with `fgets`, for the backtracking engine and for backtracking with path merging. The figure 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 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` program contains several other decision nodes with two satisfiable branches.

The sum analysis runtimes for the two test sets are given in table I. For the `memcpy` overflows path merging increases the runtime a little bit due to the overhead of computing and comparing reduced local contexts. Most of the `memcpy` programs do not contain a single decision for which both branches are satisfiable, and therefore no merge possibilities. The `fgets` test programs all contain such decisions, and the sum runtime is reduced by path merging from 80.7 s to 34.4 s. The sum runtime for the 54 programs without merging is 94 s, while path merging reduces it to 50s. The overall speedup with path merging on the test set is therefore about two, which is considerable for the tiny Juliet programs.

V. Related Work

There is a large body of work on symbolic execution available which spans over 30 years [10]. Dynamic symbolic execution for test case generation for x86 binaries is presented in [20]. To reduce complexity, only variables are modelled as symbolic which directly depend on program input, in order to find exploitable bugs. Most tools perform symbolic execution on an intermediate code representation. Apart from [6], where LLVM intermediate code is analyzed using a worklist algorithm, prominent symbolic execution engines are presented in [21] and [22]. In [21], dynamic symbolic execution of the Common Intermediate Language (MSIL/CIL) is performed for test case generation. The engine described in [22] analyzes Java bytecode. Sound path merging based on dead path differences is presented in [15], the implementation extends [6]. Merging of paths with live differences is investigated in [23]. 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 [6].

VI. Conclusion

This paper described the extension of a source-level backtracking symbolic execution engine for C/C++ with path merging functionality and its implementation in Eclipse CDT. The evaluation with tiny test programs from the Juliet suite already showed 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 extensions in different directions. One is to investigate the effect of additional merge points, for example at connector nodes after `if`/`else` and `switch` statements and loops, A memory-efficient implementation of the context cache might exploit redundant information due to shared sub-paths. The very simple live variable analysis implementation can be improved to find more merge possibilities. Inter-procedural live variable analysis could find merge possibilities, e.g., in certain flow variants with dead global variables. Another direction is the extension to support path merging in the analysis of multi-threaded code, in a straight-forward combination with [24]. A way to make the analysis scalable in order to analyze practical programs is to restrict the code coverage, for example, to branch coverage. There are less merge possibilities when coverage is restricted to fewer program paths, but path merging remains applicable without changes.

Acknowledgement

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

References

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

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

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

[4] 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.

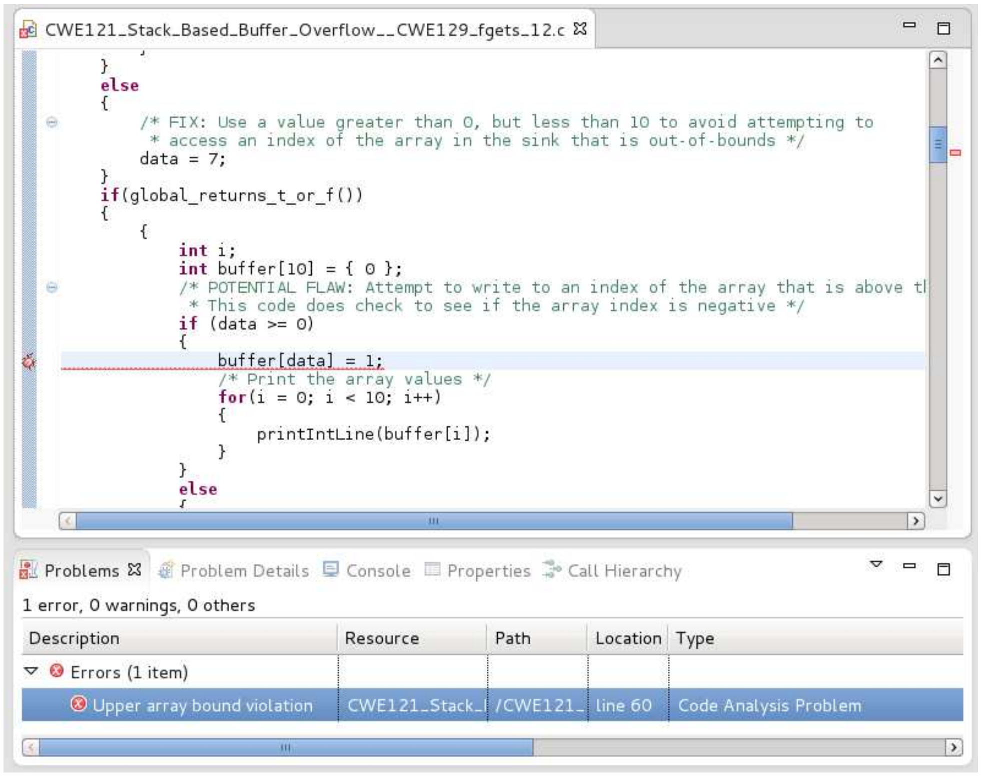
[5] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically generating inputs of death," in 13th ACM Conference on Computer and Communications Security (CCS), 2006, pp. 322–335.

[6] 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.

[7] 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.

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

[9] 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.


CWE121_Stack_Based_Buffer_Overflow__CWE129_fgets_12.c

```
    }
    else
    {
        /* FIX: Use a value greater than 0, but less than 10 to avoid attempting to
         * 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 above tl
             * 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
```

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
Upper array bound violation	CWE121_Stack_	/CWE121_	line 60	Code Analysis Problem

Figure 7. Error reporting in the Eclipse GUI.

[10] 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.

[11] T. Kremenek, “Finding software bugs with the Clang static analyzer,” LLVM Developers’ Meeting, Aug. 2008, retrieved: 09/2014. [Online]. Available: http://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf

[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] A. Laskavaia, “Codan- C/C++ static analysis framework for CDT,” in EclipseCon, 2011, retrieved: 09/2014. [Online]. Available: http://www.eclipsecon.org/2011/sessions/index0a55.html?id=2088

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

[15] 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.

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

[17] 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.

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

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

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

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

[22] 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.

[23] 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.

[24] A. Ibing, “Path-sensitive race detection with partial order reduced symbolic execution,” in Workshop on Formal Methods in the Development of Software (WS-FMDS), 2014, in press.