Dynamic Symbolic Execution using Eclipse CDT

Andreas Ibing

Chair for IT Security
TU München
Boltzmannstrasse 3, 85748 Garching, Germany
Email: andreas.ibing@tum.de

***Abstract*—Finding software bugs before deployment is essential to achieve software safety and security. The achievable code coverage and input coverage with manual test suite development at reasonable cost is limited. Therefore, complementary automated methods for bug detection are of interest. This paper describes automated context-sensitive detection of software bugs with dynamic symbolic execution. The program under test is executed in a debugger, and program execution is automatically driven into all program paths that are satisfiable with any program input. Program input and dependent data are treated as symbolic variables. Dynamic analysis and consistent partial static analysis are combined to automatically detect both input-dependent and input-independent bugs. The implementation is a plug-in extension of the Eclipse C/C++ development tools. It uses Eclipse's code analysis framework, its debugger services framework and a Satisfiability Modulo Theories automated theorem prover. The resulting dynamic symbolic execution engine allows for consistent partially concrete program execution. Compared to static symbolic execution, it transfers as much work as possible to concrete execution in a debugger, without losing bug detection accuracy. The engine is evaluated in terms of bug detection accuracy and runtime on buffer overflow test cases from the Juliet test suite for program analyzers.**

Keywords–Testing; Program analysis; Symbolic execution.

I. INTRODUCTION

Software bugs in general are context-sensitive, so that a context-sensitive algorithm is needed for accurate detection. Symbolic execution [1] is an approach to automated context-sensitive program analysis. In can be applied as static analysis, in the sense of symbolic interpretation. Program input is treated as symbolic variables, and operations on variables yield logic equations. The satisfiability of program paths and bug condition satisfiability are decided with an automated theorem prover (constraint solver). Symbolic execution in principle is applicable to all levels of software, i.e., models, source code, intermediate code and binaries.

As code execution is in general faster than interpretation, symbolic execution has also been applied as dynamic analysis [2]. When certain software parts can not practically be treated by static symbolic execution, this offers a way for dynamic analysis by concretizing symbolic variables as approximation, without introducing false positive bug detections (although this leads to false negative detections) [2]. The software parts which are treated symbolically (static) and which concretely (dynamic) can be made selectable for a tool user in the sense of selective symbolic execution [3]. A more detailed overview of available symbolic execution tools and applications is available in [4][5].

Symbolic execution relies on an automated theorem prover as logic backend. The current state in automated theorem proving are Satisfiability Modulo Theories (SMT) solver [6]. An example state of the art solver is described in [7]. A standard interface to SMT solvers has been defined with the SMTlib [8].

The different types of softwate bugs are classified in the common weakness enumeration (CWE) [9]. Examples are stack based buffer overflows with number CWE-121 and heap based buffer overflows as CWE-122.

For the evaluation of automated software analyzers, test suites have been developed. The evaluation criteria are the number of false positive and false negative bug detections and the needed run time. Currently, the most comprehensive test suite for C/C++ is the Juliet suite [10]. It systematically tests the correct detection of different common weakness types (as baseline bugs) in combination with different data and control flow variants which cover the available programming language grammar constructs. The suite contains both 'good' (without bug) and 'bad' (including a bug) functions in order to measure false positive and false negative detections. The maximum bug context depth of a flow variant are five functions in five different source files.

This paper describes a dynamic symbolic execution approach, that combines static and dynamic checks in order to detect both input-dependent and input-independent bugs. It builds upon an existing purely static symbolic execution engine described in [11]. The work at hand differs in that most of the work is transferred to a debugger, which additionally allows for consistent partially concrete program execution. The debugger is automatically driven into all executable program paths, and bugs are detected both during concrete execution and during symbolic interpretation.

The remainder of this paper is organized as follows: Section III gives an overview of the algorithm which is used to traverse the program execution tree. Section IV describes the implementation which extends the Eclipse C/C++ development tools (CDT). In Section V the achieved bug detection accuracy and run times are evaluated with buffer overflow tests from the Juliet suite. Section II gives an overview of related work, and Section VI discusses the obtained results.

II. RELATED WORK

There is a large body of work on symbolic execution available which spans over 30 years [12]. Dynamic symbolic execution is described in [2][13][14][15]. To reduce complexity and increase analysis speed, as many variables as possible

 ISBN: 978-1-61208-438-1

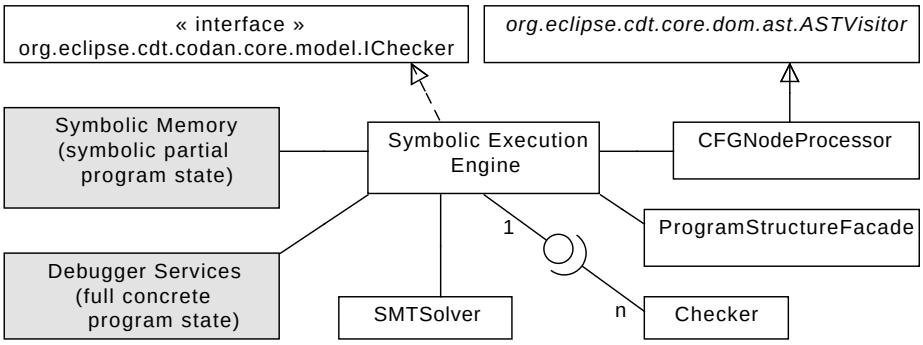

Figure 1. Architecture overview.

are regarded as concrete values. Only variables which 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 [13], single-stepping is used together with a check whether a symbolic variable is contained in the respective statement. Selective symbolic execution is presented in [3]. 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 [15] on it.

The presented approach differs in several aspects. One point is the tight IDE integration which might enable synergies with other Eclipse tools. Static and dynamic checks are combined in order to also detect bugs which are not input dependent. Breakpoints are set adaptively to interprete as few code lines as possible without degrading bug detection accuracy, and complex dependencies between symbolic variables are tracked.

III. Algorithm

The algorithm is basically depth-first search. It is used to traverse the tree of satisfiable paths through the program under test, which is commonly called the program execution tree.

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 contains a full conrete program state. The interpreter contains the partial variable set, which needs to be symbolic, i.e., the values are logic formulas. The full concrete program state and the partial symbolic state are consistent, i.e., the concrete state satisfies the symbolic state constraints.

C programs interact with their environment through functions from the C standard library (libc). The symbolic execution engine can trace input and can determine 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. Breakpoints are inserted only in the source files of interest.

For the first execution path, the engine traces program input. In case of blocking functions, a direct return with a valid error return value is forced. The argument vector is initially set to be empty.

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.

Bug detection in concretely executed program parts is performed with run-time checks, i.e., dynamic analysis. Bug detection in interpreted parts, i.e., input dependent, is performed using satisfiability queries to the solver.

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.

IV. Implementation as Eclipse CDT plug-in

A. Architecture Overview

An overview of the architecture is given in Figure 1 as class diagram. The symbolic execution engine performs tree-based interpretation [16] for program locations which use symbolic variables. The engine can be started from the CDT GUI through an extension point provided by the code analysis framework. The syntax files of interest are parsed into abstract syntax trees (AST) with CDT's C/C++ parser. Translation into logic equations is performed according to the visitor pattern [17] using CDT's `ASTVisitor` class. 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. A full concrete program state is available in the debugger. For the detection of input dependent bugs, 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.

B. Short review of CDT's code analysis framework

The code analysis framework (Codan [18]) is a part of CDT. It provides GUI integration for checker configuration

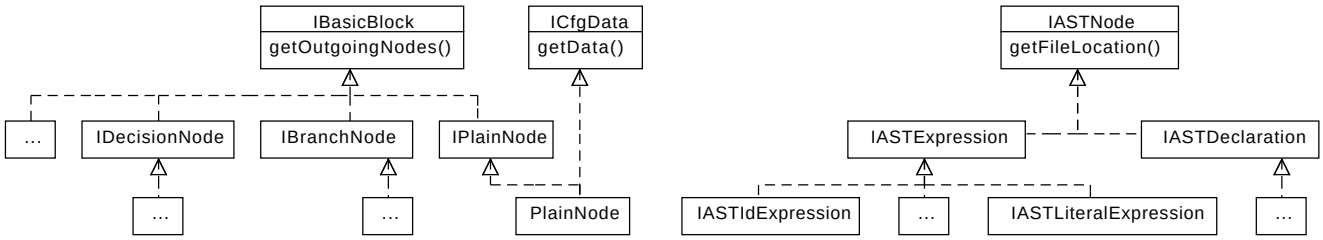

Figure 2. Important data structures provided by CDT and its code analysis framework for CFG (left) and AST (right).

and result presentation using Eclipse's marker framework. It further provides a control flow graph (CFG) builder. Figure 2 illustrates important data structures for AST and CFG. 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. Static program analysis normally evaluates paths through CFGs [11]. Tree-based interpretation means that for a CFG node the referenced AST subtree is interpreted. An AST node provides a reference to the corresponding source location.

C. Short review of 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) [19], which is an abstraction layer over debuggers' machine interfaces. DSF provides a set of asynchronous services. The main service interfaces are illustrated in Figure 3. They are 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`).

D. Partial symbolic interpretation

The debugger stops at breakpoints or when it 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 CFG node resolution, a location map for the source files of interest is pre-computed before analysis start. The resolved CFG node is then followed to its AST subtree, which is symbolically interpreted. Needed concrete values are queried from the debugger The translation into logic equations uses the SMTlib sublogic of arrays, uninterpreted functions and bit-vectors (AUFBV).

E. Input-dependent branches

The debugger only breaks at a decision when the decision contains a symbolic variable (input-dependent branch). 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 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.

F. 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 [20]. 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 pre-computed 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 3).

For accurate bug detection it is additionally necessary to trace pointer targets. Pointer assignments must be traced because there may be pointers to a target when the target becomes symbolic. The initial breakpoints therefore also include breaks on all pointer assignments (a pointer target map is then updated accordingly). This requirement is illustrated in the experiments section with Figure 5.

G. Controlling program input

In order to trace and force program input, the debugger is set to break at calls to functions from the standard library. Because the `libc` contains several functions for which the debugger cannot step into or break inside (e.g., functions that directly access the virtual dynamically shared object), these functions are wrapped. For non-blocking functions on the first program path, the `step into` and `finish` debugger commands are used to trace the function's return value. For blocking functions or on later program paths, the program input is set to the pre-determined value (or solver-generated value respectively) with `step into` and `return` debugger commands.

H. Bug detection

1) During concrete execution: Instrumentation and dynamic analysis are used to detect bugs during concrete execution. This paper uses the example of buffer overflow detection. To dynamically detect buffer overflows, the available address sanitizer from [21] is used. In [21], it is reported that the instrumentation slows down execution by about 73% and

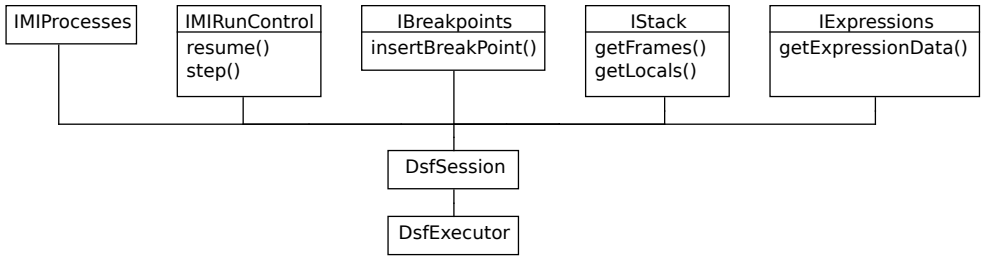

Figure 3. Main interfaces of CDT's debugger service framework.

```
1  void CWE121_memcpy_01_bad() {
2    charvoid cv_struct;
3    cv_struct.y = (void *)SRC_STR;
4    // FLAW: overwrite the pointer y
5    memcpy(cv_struct.x, SRC_STR,
6      sizeof(cv_struct));
7    cv_struct.x[(sizeof(cv_struct.x)/
8      sizeof(char))-1]='\0';
9    printLine((char *)cv_struct.x);
10   printLine((char *)cv_struct.y);
11 }
```

Figure 4. Buffer overflow detection during concrete execution. Example from [22]

increases memory usage about 3.4 times. The source code under test is compiled and statically linked with the address sanitizer library. A breakpoint is set on the address sanitizer error report function. In case the debugger breaks at this location, the bug is localized by following the call stack back into the source files of interest.

2) During symbolic interpretation: Input dependent bugs are detected with solver queries during symbolic interpretation. For buffer overflows, the bounds checker is triggered during interpretation of array subscript expressions and pointer dereferences, when the index expression or pointer offset are symbolic. The checker then queries the solver whether index/offset can be smaller than zero or lerger than the buffersize.

I. Input generation

The parts of an execution path that are symbolically interpreted can be denoted as symbolic execution path. 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 (`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 (and therefore also the execution tree) is complete when further backtracking is not possible.

```
1  void CWE121_fgets_32_bad() {
2    int data = -1;
3    int *data_ptr1 = &data;
4    int *data_ptr2 = &data;
5    { int data = *data_ptr1;
6      char input_buf[CHAR_ARRAY_SIZE] = "";
7      if (fgets(input_buf, CHAR_ARRAY_SIZE,
8        stdin) != NULL)
9      {  data = atoi(input_buf); }
10     else
11     { printLine("fgets() failed."); }
12     *data_ptr1 = data;
13   }
14   { int data = *data_ptr2;
15     int buffer[10] = { 0 };
16     if (data >= 0) {
17       // FLAW: possible buffer overflow:
18       buffer[data] = 1;
19       for(int i = 0; i < 10; i++)
20       { printIntLine(buffer[i]); }
21     }
22     else
23       { printLine("ERROR: out of bounds");}
24   }
25 }
```

Figure 5. Buffer overflow detection during symbolic interpretation. Example from [22]

V. Experiments

A. Test cases and test set-up

The used test set consists of 58 small buffer overflow test programs from the Juliet suite [22]. The test programs are analysed with the Eclipse plug-in, using Eclipse version 4.5 (CDT 8.8.0) on a i7-4650U CPU, on 64-bit Linux kernel 3.16.0 with GNU debugger gdb version 7.7.1 [20]. The test set contains stack based buffer overflows with `memcpy()` (19 test programs) and with `fgets()` (39 test programs). The test programs cover all 39 Juliet flow variants for C. The results are illustrated in Figure 6. The figure contains run-times for correct detection only (no false positives or false negatives). Flow variants in [22] are not numbered consecutively in order to allow for later insertions.

The test set contains bugs which are detected during concrete execution (`memcpy()`) and with input-depending branching (e.g., flow variant 12). It also contains input-depending bugs (with `fgets()`), which are detected during symbolic interpretation using the solver (bug condition satisfiability check).

 ISBN: 978-1-61208-438-1

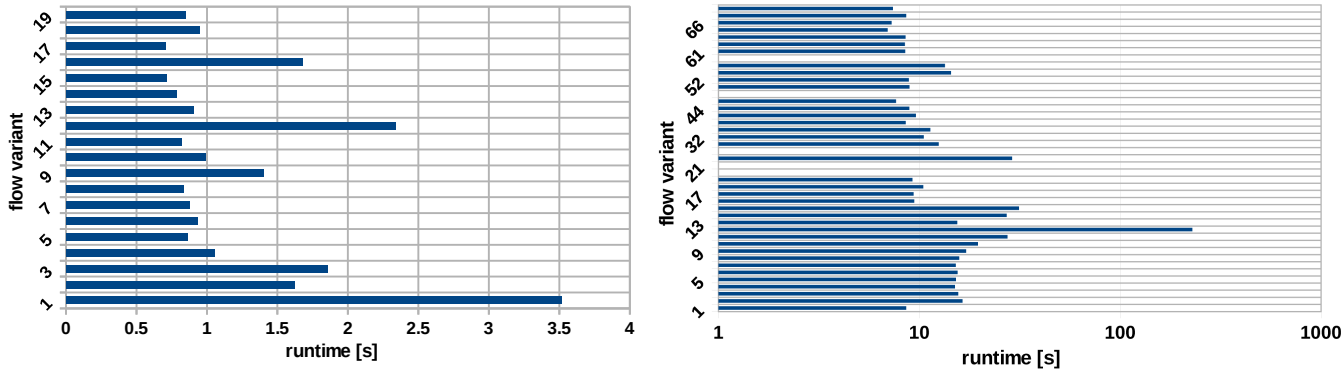

Figure 6. Analysis run-times for correct bug detection with dynamic symbolic execution.

B. Detection accuracy

The functionality is illustrated with two source code listings from [22]. The listings have been slightly modified to be shorter.

Figure 4 shows an example of bug detection during concrete execution in the debugger. 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 sanitizer's error reporting function and the bug is correctly localized.

Figure 5 illustrates the need for pointer tracing with flow variant 32 ('data flow using two pointers to the same value within the same function' [22]). 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 trigged 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.

Bug detection during concrete execution depends on the available instrumentation and run-time checks. The address sanitizer used as example first misses a buffer overflow with flow variant 9 ('control flow depending on global variables'), but then detects it through reception of a segmentation fault signal from the operating system.

The bugs are correctly detected for all flow variants apart from variant 21 ('control flow controlled by value of a static global variable'). For this variant, the CFG builder misclassifies a branch node as dead node. This leads to missing program paths in the analysis and consequently to a false negative detection.

C. Speed

Figure 6 shows analysis runtimes for the buffer overflow test cases with `memcpy` on the left, and with `fgets` on the right. The vertical axis shows the numbering of data and control flow variants from Juliet [22]. The horizontal axis shows the measured runtime (wall-clock time) in seconds. The tool needs about 1-2s for each of the `memcpy` test cases, and about 20s for each of the `fgets` test cases. An exception is the `fgets` test with flow variant 12. It contains quite a few concatenated decisions for which both branches are satisfiable. This leads to exponential path explosion. In addition, the debugger execution is restarted many times from the program start. This means that overlapping start paths are re-executed redundantly.

VI. Conclusion and Future Work

This paper presents an Eclipse CDT plug-in for automated bug detection with dynamic symbolic execution. Software bugs are detected with combined static and dynamic checks. As much work as possible is transferred to a debugger, whose execution is driven into all executable program paths. The presented approach is applicable with native and with cross compilation. It can be applied, e.g., with the qemu virtual machine monitor which contains a gdb server for the guest virtual machine. The current implementation suffers from the path explosion problem, i.e., the number of satisfiable paths in general grows exponentially with program length. Ongoing work therefore aims to improve the scaling behaviour by implementing ways to detect and prune program paths, on which the detection of new bugs is not possible.

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

[3] 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, pp. 265–278.

 ISBN: 978-1-61208-438-1

[4] 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., 2011, pp. 1066–1071.

[5] 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.

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

[7] ——, “Z3: An efficient SMT solver,” in Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 337–340.

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

[9] 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.

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

[11] A. Ibing, “Symbolic execution based automated static bug detection for Eclipse CDT,” Int. J. Advances in Security, vol. 1&2, 2015, pp. 48–59.

[12] 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.

[13] 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.

[14] P. Godefroid, M. Levin, and D. Molnar, “Automated whitebox fuzz testing,” in Network and Distributed System Security Symp. (NDSS), 2008, pp. 151–166.

[15] 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.

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

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

[18] E. Laskavaia, “Codan- a C/C++ code analysis framework for CDT,” in EclipseCon, 2015.

[19] P. Piech, T. Williams, F. Chouinard, and R. Rohrbach, “Implementing a debugger using the DSF framework,” in EclipseCon, 2008.

[20] R. Stallman, R. Pesch, and S. Shebs, “Debugging with gdb,” 2011, [retrieved: Sept., 2015]. [Online]. Available: http://sourceware.org/gdb/current/onlinedocs/gdb.html

[21] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Address-Sanitizer: A fast address sanity checker,” in USENIX Annual Technical Conference, 2012, pp. 28–28.

[22] Juliet Test Suite v1.2 for C/C++, United States National Security Agency, Center for Assured Software, May 2013, [retrieved: Sept., 2015]. [Online]. Available: http://samate.nist.gov/SARD/testsuites/juliet/Juliet_Test_Suite_v1.2_for_C_Cpp.zip

 ISBN: 978-1-61208-438-1