SMT-Constrained Symbolic Execution Engine for Integer Overflow Detection in C Code

Paul Muntean, Mustafizur Rahman, Andreas Ibing, and Claudia Eckert
Technische Universität München, Department of Informatics
Boltzmannstraße 3, 85748 Garching, Germany
Email: {paul, mustafizur, ibing, eckert}@sec.in.tum.de

***Abstract*—Integer overflow errors in C programs are difficult to detect since the C language specification rules which govern how one can cast or promote integer types are not accompanied by any unambiguous set of formal rules. Thus, making it difficult for the programmer to understand and use the rules correctly causing vulnerabilities or costly errors. Although there are many static and dynamic tools used for integer overflow detection, the tools lack the capacity of efficiently filtering out false positives and false negatives. Better tools are needed to be constructed which are more precise in regard to bug detection and filtering out false positives. In this paper, we present an integer overflow checker which is based on precise modeling of C language semantics and symbolic function models. We developed our checker as an Eclipse plug-in and tested it on the open source C/C++ test case CWE-190 contained in the National Institute of Standards and Technology (NIST) Juliet test suite for C/C++. We ran our checker systematically on 2592 programs having in total 340 KLOC with a true positive rate of 95.49% for the contained C programs and with no false positives. We think our approach is effective to be applied in future to C++ programs as well, in order to detect other kinds of vulnerabilities related to integers.**

***Index Terms*—software vulnerability, information security, context-sensitive analysis**

I. Introduction

In the 2011 top 25 most dangerous software errors [22] MITRE classifies integer overflows as the main source for different types of software vulnerabilities. Integer overflow errors are responsible for a series of vulnerabilities in the OpenSSH [23] and Firefox [24], both allowing attackers to execute arbitrary code. Software failures can materialize in reality as for example during the crash of the Ariane 5 flight 501 in 1996 due to an attempt to cast a floating point value to a 16-bit integer value which resulted in a truncation error.

Integer numerical errors in software applications are costly, hard to detect—there are a couple of types of integer overflows and some of them are inserted into code intentionally and others unintentionally—and exploitable. According to Brumley and colleagues [5] there are four types of integer related problems: (1) overflow (occurs at run-time when the result of an integer expression exceeds the maximum value for its respective type), (2) underflow (appears at run-time when the output of an integer expression is less than the minimum value that the assignee can hold, thus "wrapping" to the maximum integer for the type), (3) signedness (occurs when a signed integer is interpreted as unsigned, or vice-versa) and (4) truncation (appears when an integer with smaller width—number of bits—has to hold an integer with larger width). Furthermore, there are intentional or unintentional uses of integer overflows and illegal uses of operations such as shifts which could lead to integer related errors.

Typical indirect integer (integer related problems are typically exploited indirectly, contrary to e.g., buffer overflows which can be exploited directly or indirectly) bug exploitations are: (a) Denial of Service (DoS) attacks where the exploit causes infinite loops or excessive memory allocation, (b) arbitrary code can be executed when an integer vulnerability results in insufficient memory allocation which afterwards can be exploited by buffer overflows, heap overflows and overwrite attacks, (c) an upper bound sanitization check can be bypassed when unexpected negative integer values are used, (d) a logic error, where a reference counter in the NetBSD OS (CVE-2002-1490) [25] was manipulated by an attacker that resulted in the premature freeing of an object from memory and (e) array index attacks are caused by a vulnerable integer value which can be used as array index so that attackers can precisely overwrite arbitrary bytes in memory.

These types of integer related problems can be addressed with code analysis techniques (e.g., static and dynamic code analysis), formal modeling of C semantics, Extended Static Checking (ESC)—usage of code annotations, Satisfiable Modulo Theories (SMT) solver, test cases to trigger the bug, compiler integration (Clang), formal modeling of integer typing rules, etc. Typical dynamic analysis have low overhead $\approx$5%, low true positives and false negatives rate, reduced path coverage, etc. On the other side static analysis offers environment models, bit precision, low number of false positives, high path coverage, etc.

In this paper we address integer overflow vulnerabilities through precise symbolic "modeling" of C language semantics which are responsible for integer overflows and C function models. Concretely, we extended the C statement processing component of our engine and carefully remodeled each external (C standard library and any other used API) used function through usage of symbolic function models. Thus, providing a precise modeling of C language semantics is the key for a high false positives and low false negatives rate.

In summary, we make the following contributions: (i) Precise symbolic modeling of C related semantics needed for integer overflow detection, §III. (ii) An integer overflow checker Eclipse plug-in based on our static execution engine, §IV and automated testing based on automatically generated jUnit test cases and Eclipse projects, V-B and V-C. (iii) Experimental evaluation of our approach on the open source C/C++ test case CWE_190_Integer_Overflow [33], §V.

978-1-4799-7755-0/15/$31.00

The remaining paper is organized as follows: §III presents the design and architecture of our engine. §IV presents implementation details of our integer overflow checker. §V contains the evaluation of our approach. §II presents related work. Finally, §VI contains the discussion and future work.

II. Related Work

Research on detecting integer problems focuses primarily on integer overflow vulnerabilities, either employing dynamic or static code approaches [10] which run on binary or source code. The static approaches are mostly based on an analysis framework and added run-time checks at certain interesting points in code (e.g., assignments, $x := expr.$) located on satisfiable program paths.

The static analysis tool UQBTng [35] decompiles the binary files and then uses model checking based on CBMC [9] to detect integer overflows. IntScope [34] first transforms the analyzed binaries into an intermediary representation (IR) and based on symbolic execution and taint analysis it checks for integer overflows. These two approaches operate on binary files and can not figure out the original variables data types since these get lost during the compilation process whereas our approach can use the original variables data types from source code. ARCHERR [8] can examine million lines of source code but can not deal with string operations and has on average 35% of false positives whereas our approach has a lower false positives rate. Microsoft's PREfast [21] is used during source code compile time and relies on source code annotations which is integrated in the Microsoft VS IDE and are provided in advance. Our approach does not require costly annotations. On the other hand we think that the annotations represent an useful source of information which is typically not available during static analysis. Microsoft's PreFix [21] is based on the Z3 [21] solver, runs on large legacy C/C++ source code repositories in order to find a wide range of problems related to integers and uses a ranking mechanism to filter out false positives. Our tool compared to PreFix focuses on only one type of integer related problems (integer overflows) by using C function models whereas the PreFix tool does not use function models. Ceesay and colleagues [6] added type qualifiers to detect integer overflow problems. Their approach relies on expensive system extensions which are linked into the compiler and are used to check the code for integer errors. Their approach requires user annotation whereas our approach does not require annotations. Ashcraft et al. [2] and Sarkar et al. [30] used bounds checking and taint analysis to see if untrusted values are used in trusted sinks. These two approaches are based on insensitive information flow whereas our approach is context sensitive.

The dynamic analysis tools are used to detect integer related problems: RICH [5], BRICK [7], SmartFuzz [26], SAGE [14] and IOC [11]. RICH [5] instruments programs to detect safe and unsafe operations based on well-known sub-typing theory. BRICK [7] detects integer overflows in compiled executables using a modified Valgrind [28] version. Its accuracy and efficiency depend on the test input used to exercise the instrumentation. It is either slow (50× slowdown) or has many false positives. SmartFuzz [26] is also based on Valgrind, but it uses dynamic test generation techniques to generate inputs, leading to good test coverage. SAGE [14] uses dynamic test generation, but it targets fewer integer problems than SmartFuzz. IOC [11] is an integer overflow and underflow problems detection tool integrated with the Clang compiler. Compared with our approach these tools can neither exercise all bugs due to dynamically generated test cases nor full path coverage can be achieved.

III. Engine Design and Architecture

Figure 1 presents the engine architecture [18] on which our integer overflow checker is based. Starting point for the design of our integer overflow checker is the multi-threaded symbolic execution engine with backtracking described in [18], which can analyze multi-threaded C programs. It performs inter-procedural analysis and is implemented according to the tree-based interpreter pattern [29]. The engine implementation is multi-threaded which means that control flow graphs and syntax trees are shared between worker threads. It relies on a SMT solver as logic backend and translates C code into SMT-Lib [3] logic equations in the logic of Arrays, Uninterpreted Functions, Non-linear Integer and Real Arithmetic (AUFNIRA). Figure 1 contains the main classes of our engine and the interface IChecker, which makes the plug-in usable from CDT's code analysis framework [4]. Several Workers concurrently explore different parts of a program's execution tree. Each worker has an Interpreter together with a memory system model to store and retrieve symbolic variables (whose values are logic SMT-Lib equations). The translation of Control Flow Graph (CFG) nodes into SMT-Lib syntax is performed by the StatementProcessor (which extends CDT's abstract syntax tree visitor class) according to the visitor pattern [13]. The BranchValidator detects unsatisfiable branches in a program path with the help of the SMTSolver. WorkPool is used as synchronization object between the Workers and the WorkPoolManager.

We briefly present the main classes of our engine denoted with capital letters. For a more detailed description see [17]. WorkPoolManager implements the interface IChecker which is present in the Codan API. The WorkPoolManager starts workers and reports found errors through the Codan interface to the Eclipse marker framework. ProgramStructureFacade provides access to control flow graphs. WorkPool is used as synchronization object (synchronized methods) which is used to track the number of active workers and to exchange split paths. Worker has a forward and a backward (backtracking) mode which passes references to control flow graph nodes for entry (forward mode) or backtracking to the Interpreter. Interpreter follows the tree- based interpreter pattern [29]. SMT syntax is generated by the StatementProcessor (which implements CDT's ASTVisitor) by bottom-up traversal of Abstract Syntax Tree (AST) sub-trees (visitor pattern), which are referenced by CFG nodes. Symbolic variables are stored in and retrieved from MemSystem. The interpreter further offers an interface to BranchValidator and to checker classes. SMTSolver wraps the interface to the Z3 [12] external solver. BranchValidator is triggered when entering a branch node and generates a smtlib query for the path constraint. For an unsatisfiable branch it throws an exception which is further on caught by the worker. Environment provides symbolic models of standard library functions. StatementProcessor extends the ASTVisitor class contained in the Codan API. In this class each statement contained on an execution path is visited in order to create our symbolic variables and the SMT constraint system

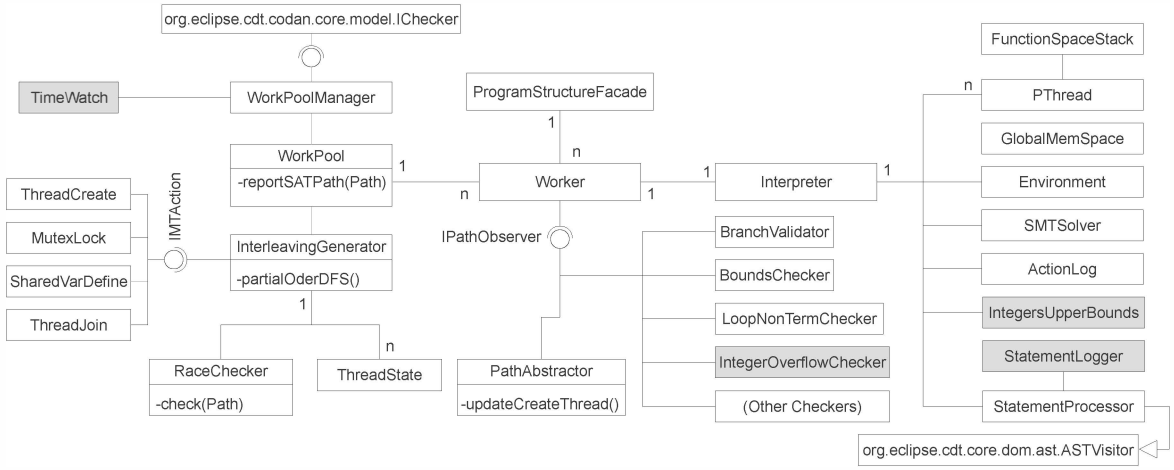

Fig. 1: Main engine classes (presented in more detail in [17])

which is attached to each symbolic variable. Among the ”leave methods”—which visit each statement AST in bottom-up order—contained in the StatementProcessor we extended the functionality of the IASTDeclaration, IASTDeclSpecifier and IASTDeclarator methods. BoundsChecker, a buffer overflow checker which triggers on memory access with (symbolic) pointers, forms bounds violation satisfiability queries and reports buffer overflows, underflows etc. LoopNonTermChecker is an infinite loop detection checker [16]. FunctionSpaceStack used to model the function space stack of symbolic functions. GlobalMemSpace modeling the global memory space of symbolic variables. RaceChecker is used for detecting race conditions [15].

The new classes which were added to support the detection of integer overflows in source code are shaded grey in figure 1. The IIntegerOverflowObserver (for the sake of brevity not depicted in figure 1) interface is extended to notify the checker about integer overflows. The interface is implemented by IntegerOverflowChecker which is described in detail in section IV-A. The IntegerOverflowChecker is used to trigger a bug report if an integer overflow has been detected. If a satisfiable path through the analyzed code is detected, then additional checks are performed inside IntegerOverflowChecker. Further on, if the resulted SMT-Lib system is still satisfiable then a bug report is generated indicating that an integer overflow error was detected.

Due to paper space limitations we will briefly describe the main features of our engine which are relevant to the integer overflow checker implementation.

1) Unrestricted context depth: Inter-procedural path-sensitive analysis with a call string approach [19], [31] is supported by the symbolic execution engine. The path sensitivity is based on per function control flow graphs with no in-lining. The function call context is represented by a program path leading to its call. The symbolic execution can be constrained regarding how many times it should run by setting a context bound (e.g., number of loop iterations) (which in general incurs accuracy degradation) or unconstrained which can lead to the possibility of non-termination (e.g., endless loops).

2) Finding program paths: A fixed deterministic thread scheduling algorithm is used for running the symbolic execution. The algorithm depends on the thread identity numbers. Lowest Thread-ID First (LTIF) scheduling is used which is based on scheduling one of the active threads having the lowest thread-ID first. The symbolic execution is run with approximate path coverage which uses Depth-First Search (DFS). During DFS program states are backtracked and branch decisions are changed, compare [18]. The loop iteration bound can be configured either to prune a path until the loop iteration bound is reached, or to bypass the loop by avoiding the BranchValidator check.

3) Automatic slicing: In order to keep the equation systems for satisfiability checks small, only relevant logic equations are passed to the solver for a certain verification condition. This corresponds to automatic slicing [32] over the control flow (for separate analysis of different program paths) and over the data flow (for verification conditions on a program path).

4) Context sharing for different checkers: With one enumeration of satisfiable paths all checks can be performed, and any specific checkers is allowed to share the contexts because it is separated from the symbolic path interpretation. The checkers are allowed through an interface to register for notifications (triggers) and to query context equations. The symbolic interpreter is queried whenever triggered in order to resolve the dependencies of the variables at the triggered location into the relevant equation system slice and adds the verification condition formula for a satisfiability query.

5) Logic representation: The SMT-Lib sub-logic of arrays, uninterpreted functions and non-linear integer and real arithmetic (AUFNIRA) has been chosen for using high-level logic that can decide automatically. With a target and a symbolic integer as offset formula, pointers are handled as symbolic pointer by the interpreter. They are outputted as logical formulas when dereferenced. Symbolic variables are created for the fields of composite data structures (e.g., structs and in case of C++ also classes) and are not translated, but rather treated like scopes.

6) Path validation: PathValidator is triggered for branch nodes and uses the same interface as checkers. For all path

decisions up to the current branch the PathValidator queries the equation SMT-lib linear system slice based on the resolution of the variable dependencies. Next, it adds a satisfiability check. The PathValidator throws a PathUnsatException if the solver answers unsatisfiable, which is caught by the PathExplorer(which reports the un-satisfiable path to the PathValidator and symbolic execution proceeds with the next path).

7) SMT Solving: The common Eclipse distributions come with a SAT solver plug-in [20], a SMT solver plug-in is unfortunately not (yet) available. Therefore the SMT solver Z3 described in [12] is used. It is wrapped by the SMTSolver class and started as external process.

8) Eclipse extension: The WorkPoolManager implements the Codan IChecker interface by pluging in the extension point "org.eclipse.cdt.codan.core.checkers". While the available Codan checkers are normally configured to be "run as you type" or "run with build", the symbolic execution engine is only "run on demand" with a GUI command, because of higher complexity and larger run-time of path-sensitive analysis. The plug-in further uses Codan ControlFlowGraphBuilder to generate CFGs for parts of an AST which are rooted in a function definition.

IV. Implementation

A. Integer Overflow Checker Implementation

Our integer overflow checker is notified from inside the StatementProcessor when assignment statements are encountered, with a symbolic variable which includes a symbolic variable name and symbolic type. The Interpreter is notified by calling "ps.notifyLimitChecker(ini_ssa);". Next, the notification is delegated to the appropriate integer overflow checker by the Interpreter which checks if there could be an integer overflow. The slice of equations on which the "ini_ssa" (this is a symbolic variable used to statically model the run-time variable x, $x := expression$) variable depends is queried by the checker, and adds one satisfiability check. The check is used to verify if the symbolic variable "ini_ssa" can be greater than the used integer upper bound value (the upper bound values are extracted from the C standard library "limits.h" file). If the solver answers "SAT" (satisfiable) to the query, then the problem is reported. The bug report contains the problem ID (unique system string), file name where the bug was detected and line number where the bug is located. In principle CWE-190—integer underflow (wrap or wrap-around) and CWE-192—integer coercion error are detectable.

Any number of checkers can be added and share the symbolic execution contexts. The Codan extension point supports the addition of new problems and problem detail views. Detected problems are reported to the marker framework with their Id, file name, line number and problem description. We added our path-sensitive integer overflow checker alongside other existent checkers (RaceCondition checker, InfiniteLoop checker, etc.).

The grey shaded classes depicted in figure 1 were added, (≈1400 SLoC). In StatementProcessor (≈700 SLoC) we added code in the AST traversing leave() methods for dealing with the new types of C statements contained in the analyzed programs. IntegerOverflowChecker (105 SLoC) is triggered for variable assignments present in the analyzed code. It generates satisfiability queries used to check for violation of integer overflows and reports an error in case of satisfiability. IntegersUpperBounds (42 SLoC) was used to extract the actual values for the integer upper bounds (platform dependent) from the standard C library file "limits.h" by taking into account the current CPU architecture (32bit or 64bit). This makes our approach platform-independent. TimeWatch (22 SLoC) is an utility class used for time measurements of our checker. StatementLogger (38 SLoC) utility class was used to log statements coming from leave() methods present inside the StatementProcessor.

The symbolic function models: AbsModel(), SqrtModel() and RandModel() were added to the Environment inside our engine (not depicted in figure 1) in order to symbolically model the mathematical functions: abs(), sqrt() and rand(). The mathematical function "abs()" was symbolically remodeled by using the symbolic function model AbsModel (40 SLoC) inside our engine. We attached to the symbolic parameter variable of AbsModel(var_symbolic), a SMT-Lib constraint (it checks the numeric value of the "abs()" function parameter; if the parameter value is positive then the value will be not changed; else if the parameter value is negative then the "-" sign will be removed) which was used to model the mathematical modulus function. As symbolic return of the function model AbsModel(var_symbolic) we used a symbolic copy of "var_symbolic" which contained the previous attached SMT-Lib constraint. Next, we simulated the execution of the mathematical "sqrt" function call inside the function model SqrtModel, (54 SLoC), by attaching to the symbolic parameter variable of SqrtModel(param_symbolic) the value of the sqrt operation and assigning this to the symbolic variable, "param_symbolic". We implemented our own "sqrt" function in code which can deal with big integers based on the "java.math.BigInteger". Furthermore, we added the symbolic function model RandModel(), (29 SLoC), used to statically model the mathematical rand() function contained in the C standard library.

The symbolic function models: SocketModel(), ListenModel(), ConnectModel(), RecvModel(), AcceptModel() and BindModel() were used (not presented in figure 1) in order to symbolically model the (f) communication API functions: socket(), listen(), connect(), recv(), accept() and bind() (note, for sake of brevity parameters are not indicated) declared in "winsock2.h", "windows.h", "direct.h", "sys/types.h", "sys/socket.h", "netinet/in.h", "arpa/inet.h" and "unistd.h" The above mentioned functions were used inside "if" conditions containing the C "break;" statement inside the "then" branch. The analyzed C/C++ programs contained "if" conditions which were used to check if the return value of the functions calls: (f) are equal to SOCKET_ERROR ("-1"). In case the return value was equal to "-1" then "break;" was called on the "if" branch—an "else" branch was not present in the code. The symbolic return value of these functions would break the symbolic program execution if it would be equal to the macro SOCKET_ERROR "-1" or not initialized with a value. Thus, making the code located after this function calls not reachable with respect to the program execution paths and thus, it will not be possible to detect the integer overflow bug. We attached to the symbolic return variables of the function models: SocketModel (39 SLoC), ListenModel (39

SLoC), ConnectModel (41 SLoC), RecvModel (42 SLoC), AcceptModel (41 SLoC), and BindModel, (41SLoC) numeric values through the usage of SMTLib constraints. Thus, in this way the part of the code where the bug was located could be reached. Note, that every numeric value can be used as symbolic return value of the functions, except "-1". Furthermore, the function models HtonsModel (32 SLoC) and Inet_addrModel (32 SLoC) were added in order to model the htons() and inet_addr() functions. Each symbolical function model has a constructor method, getName(), getSignature() and an execute() method which makes the creation and usage of new function models straight forward.

V. Experiments

A. Methodology

We tested our checker with the open source integer overflow test case CWE_190_Inger_Overflow contained in the Juliet test suite [33]. The used test case contains 54 baseline programs with 48 Control Flow Variants (CFV) each resulting in total of 2592 analyzed programs and 2592 (†) true positives. Every baseline test case—48 CFV—contains 38 C programs and 10 C++ programs. First, we generated 26 jUnit test classes containing 100 jUnit test methods in each of the first 25 classes and respectively 92 jUnit test methods in the 26th class. Second, we ran each of the generated classes separately—due to Eclipse run-time limitations it was not possible to put all 2592 jUnit test methods in one class and run everything at once—by using the Eclipse jUnit testing environment. The focus of the experiment is to find out the number of false positives, false negatives, true positives (accuracy) and checker run-time timings (efficiency). We tested the integer overflow checker on the Eclipse IDE v. Kepler SR 1, OpenSUSE 13.1 OS, 64bit; 12GB RAM, CPU Q9550 2.83GHz, 64bit.

B. Automated jUnit Test Cases Generation

A script was developed to generate the jUnit test methods for 2592 analyzed C programs. Our script requires the directory location—path—as parameter followed by the Juliet test case name for which the jUnt test cases should be generated (CWE_190 in our case). The script uses a predefined jUnit template method and generates 100 jUnit methods per Java class file. For 2592 programs we obtained 26 jUnit test classes containing 100 jUnit test methods each and 92 jUnit test methods in the 26th test class. The dynamic parameters which are added in each Java source files are: the line numbers where the bug is located, method names and the class names. These are automatically generated based on the test cases names detected in the selected Juliet test case. For the class names we grouped the test programs in groups of 100 and named the Java class file as "CWE_" followed by the name of the 1st test program of the group of 100, followed by underscore "_" and the last test program name contained in the group. The first name of the file contained in each test program was used as names for the generated jUnit test methods. The line number where the true positive is located was determined by tokenizing the source files contained in each test program and by searching for the string (p) "/* POTENTIAL FLAW */"—this string was inserted in code by the Juliet test suite creators in order to mark the true positive and false positive locations. The script identifies as bug location the next line number after the string (p) was detected. Multiple appearances of (p) are filtered out by using several flags and counter variables used to count the number of appearances and the locations (line numbers) of (p).

C. Automated Eclipse C/C++ Programs Generation

A second script was created in order to generate 2592 Eclipse projects containing all analyzed C/C++ programs. The projects generation script uses the directory containing the Juliet test case (CWE_190 in our case) as parameter in order to iterate recursively through all folders of the main directory and generates Eclipse C/C++ projects with the required header files contained inside. In brief the script uses as templates the Eclipse C/C++ hidden project files ".cproject" and ".project" which were inserted in each generated project. The script replaces in each generated project the names of the project with the names extracted from the names of the source files contained in each Eclipse test case program. The script creates the project folder using the same name and puts all the required files for the current test project inside the folder (needed header files were also copied inside the folder). The project names and the appropriate project configurations are written in the hidden project description files (.cproject and .project). Next, the C code line "#define INCLUDEMAIN" is added after the code line "#ifdef INCLUDEMAIN" which is contained by default in each Juliet test case program in order to have a starting point for the static analysis.

D. Experimental Results

Note, that each number [1, 54] located on the X axis of figure 2 has two bars associated. Figure 2 depicts for each of the 54 test cases (each containing 48 CFV—in total 2592 C/C++ programs) the run-time in seconds indicated on the left Y axis (e.g., the left bar in figure 2 located on the X axis for #1) and the contribution of each CFV in % depicted on the right Y axis (e.g., the right bar in figure 2 located on the X axis for #1). The main contribution in all 54 test cases has CFV 12 depicted in figure 2 with ■CFV 12. We observe that baselines (#12, #24, #27, #36, #48 and #54) depicted in figure 2 and in table I, first column, have high execution times—more than 200 seconds—compared to the rest of the programs. In each of the expensive—have run-time over 200 seconds—test cases CFV 12 has more than 80% contribution to the checker run-time. Furthermore, we measured the total execution time with respect to successful triggering, 3638.122 seconds (3666.36 seconds − 28.238 seconds) and the total execution time without successful triggering, 28.238 seconds (3666.36 seconds − 3638.122 seconds) and found out that 0,77% (28.238 seconds out of 3638.122 seconds) additional performance overhead was induced by the programs in which no execution exception was raised and no bug was found.

Figure 2 was split—depicted with dashed lines "¦" in figure 2—from left to right having 12 test cases (24 bars) in each segment for the first 4 segments and 6 test cases (12 bars) in the last segment located at the far most right in figure 2. with the goal to depict commonalities between each of the 5 obtained segments. We observe that the execution times are rising in each segment to a peak value (segment 1 (#2), segment 2 (#18), segment 3 (#27), segment 4 (#47) and segment 5 (#54)) and then they abruptly drop, except

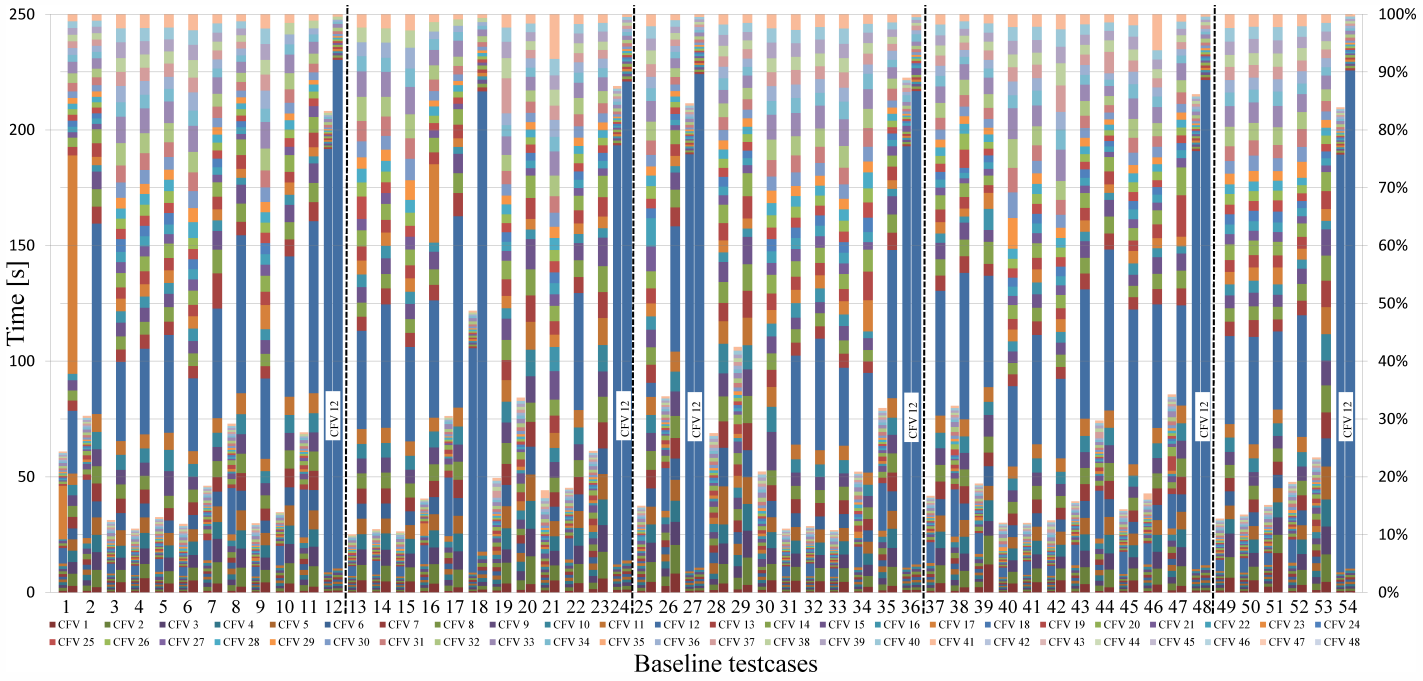

Fig. 2: Integer overflow checker run-time results for CWE-190

segment 5 where it increases continuously until it reaches the peak execution time (#54) for the last baseline test case. By dividing the whole execution time among all ”the expensive” execution baselines, (#12, #24, #36, #27 and #48) for the first 4 segments and the #54 baseline test case we observe that those 6 baselines significantly dominated the whole execution time having execution times of more than 200 seconds. The run-time for the above mentioned test cases is higher compared to other baseline test cases because these programs contain the C standard library function calls ”rand()” and ”sqrt()” in multiple nested ”if” conditions which make the path conditions to be more complex than programs which do not contain such complex nested path conditions. Thus, incurring the additional computational overhead.

Table I contains the following abbreviations: Baseline Programs (BP, contains 48 CFV), Source Lines of Code (#SLoC), Total Bugs Triggered from 48 TP (TBT), Total Execution Time in seconds (TET [s]), Total Exceptions (TE), Exceptions with Trigger (0 NO/1 YES) (EwT), Total True Positives percentage w.r.t. 48 CFV and 37 CFV (48 CFV − 11 CFV, 10 C++ programs and 1 program containing the C ”goto” statement) (TTP 48%/37%). Table I columns 5 and 8 depict the checker run-time timings in seconds and true positive percentages for 37/48 CFV, respectively. False positives and false negatives are not depicted in table I since they were not encountered during our experiment. Among the triggered bugs we have 100% success rate with respect to true positives. On the other hand, there were in total 11 (w.r.t. 2592 programs) true positives detected where we got checker run-time exceptions but the bugs have still been triggered correctly, table I column 7. However, there were in total 43 (w.r.t. 2592 programs) programs where the checker successfully parsed the source code but failed to trigger any bug—table I column 7.

Table II contains the following abbreviation: % of Detected Bugs w.r.t. to the total number of true positives 2592 (†), (% DB). Table II presents the results of running our checker on the 2592 programs. Our integer overflow checker triggered in total 1908 true positives with a success rate of 73.61% DB

TABLE I: Bug detection results for CWE_190

#	BP	#SLoC	TBT	TET [s]	TE	EwT	TTP 48%/37%
1	fscanf_add	5233	37	14.857	11	0	77.08%/100%
2	char_fscanf_multiply	5539	37	76.356	11	1	77.08%/100%
3	char_fscanf_square	5309	37	31.213	11	0	77.08%/100%
4	char_max_add	5236	37	27.566	11	0	77.08%/100%
5	char_max_multiply	5541	37	32.54	11	0	77.08%/100%
6	char_max_square	5309	37	29.631	11	0	77.08%/100%
7	char_rand_add	5236	37	46.118	11	0	77.08%/100%
8	char_rand_multiply	5541	37	72.947	11	1	77.08%/100%
9	char_rand_square	5308	37	29.996	11	0	77.08%/100%
10	int64_t_fscanf_add	5228	29	34.702	16	0	60.45%/78.37%
11	int64_t_fscanf_multiply	5493	29	69.183	17	1	60.45%/78.37%
12	int64_t_fscanf_square	5261	29	208.189	16	0	60.45%/78.37%
13	int64_t_max_add	5188	29	25.117	16	0	60.45%/78.37%
14	int64_t_max_multiply	5493	29	27.401	16	0	60.45%/78.37%
15	int64_t_max_square	5261	27	26.483	16	0	43.75%/72.97%
16	int64_t_rand_add	5188	29	40.497	16	0	60.45%/78.37%
17	int64_t_rand_multiply	5493	29	76.312	17	1	60.45%/78.37%
18	int64_t_rand_square	5261	29	121.849	16	0	60.45%/78.37%
19	int_connect_socket_add	12168	36	49.439	12	0	75 %/97.29%
20	int_connect_socket_multiply	12473	36	84.226	12	0	75 %/97.29%
21	int_connect_socket_square	12241	36	44.115	12	0	75 %/97.29%
22	int_fgets_add	6377	37	45.184	11	0	77.08%/100 %
23	int_fgets_multiply	6682	36	61.268	12	0	75 %/97.29%
24	int_fgets_square	6450	37	218.919	11	0	77.08%/100 %
25	int_fscanf_add	5188	37	37.441	11	0	77.08%/100 %
26	int_fscanf_multiply	5493	37	84.837	11	1	77.08%/100 %
27	int_fscanf_square	5261	37	211.418	11	0	77.08%/100%
28	int_listen_socket_add	13736	36	68.908	12	0	75 %/97.29%
29	int_listen_socket_multiply	14041	36	106.101	12	0	75 %/97.29%
30	int_listen_socket_square	13809	36	52.266	12	0	75 %/97.29%
31	int_max_add	5188	37	27.63	11	0	77.08%/100%
32	int_max_multiply	5493	37	28.726	11	0	77.08%/100%
33	int_max_square	5261	34	26.957	11	0	70.83%/91.89%
34	int_rand_add	5188	37	52.185	11	1	77.08%/100%
35	int_rand_multiply	5493	37	79.76	11	1	77.08%/100%
36	int_rand_square	5261	37	222.582	11	0	77.08%/100%
37	short_fscanf_add	5188	37	41.683	11	0	77.08%/100%
38	short_fscanf_multiply	5493	37	80.723	11	1	77.08%/100%
39	short_fscanf_square	5261	37	46.989	11	0	77.08%/100%
40	short_max_add	5188	37	30.207	11	0	77.08%/100%
41	short_max_multiply	5493	37	30.051	11	0	77.08%/100%
42	short_max_square	5261	35	31.186	9	0	72.92%/94.59%
43	short_rand_add	5188	37	39.372	11	0	77.08%/100%
44	short_rand_multiply	5493	37	74.297	11	1	77.08%/100%
45	short_rand_square	5261	37	35.885	11	0	77.08%/100%
46	unsigned_int_fscanf_add	5188	37	42.742	11	0	77.08%/100%
47	unsigned_int_fscanf_multiply	5493	37	85.65	11	0	77.08%/100%
48	unsigned_int_fscanf_square	5261	37	215.401	11	0	77.08%/100%
49	unsigned_int_max_add	5188	37	31.904	11	0	77.08%/100%
50	unsigned_int_max_multiply	5493	37	33.569	11	0	77.08%/100%
51	unsigned_int_max_square	5261	34	37.796	11	1	77.08%/91.89%
52	unsigned_int_rand_add	5188	37	47.831	11	1	77.08%/100%
53	unsigned_int_rand_multiply	5493	36	58.378	10	0	75 %/97.29%
54	unsigned_int_rand_square	5261	37	209.779	11	0	77.08%/100%
-	**Total**	**337.573**	**1908**	**3666.36**	**684**	**1(11) /0(43)**	**73.61% /95.49%**

TABLE II: Integer overflows bugs triggered

Triggered / 48 CFV	# of base-lines	Baseline #	# of bugs	% DB
37	34	[1, 9], 22, [24, 27], 31, 32, [34, 41], [43, 50], 52, 54	1258	48.53%
36	8	19, 20, 21, 23, 28, 29, 30, 53	288	11.11%
35	1	42	35	1.35%
34	2	33, 51	68	2.62%
29	8	10, 11, 12, 13, 14, 16, 17, 18	232	8.95%
27	1	15	27	1.04%
Total			1908	73.61%

in total as it can be observed in table II columns 4 and 5. In terms of bug triggering for the 48 CFV for each base line test case our plug-in triggered at most 37 bugs in a single baseline which corresponds to a success rate of 48.53% DB while 27 being the lowest triggering number corresponding to 1.04% DB. The rest of the DB percentages are between 1.35% and 11.11% as depicted in table II, 5th column.

TABLE III: Impact of expensive baseline test cases

Execution Times	T [s]
Total time for 12th, 24th, 36th, 48th, 54th and 27th baselines	1286.28
Average time for 12th, 54th and 27th	214.38
Execution time excluding higher value	2380.07
Average execution time excluding higher value	49.58
Average execution time per baseline programs (48 CFV)	67.89
Total execution time	3666.36

Table III shows the impact of expensive baselines test cases with respect to the total execution time, 3666.36 seconds. The average execution time per baseline test case is 67.90 seconds but if we exclude those 6 expensive baselines then the average execution time decreases by almost 20 seconds to 47.00 seconds. In our experiments we summed up the times where we had run-time exceptions with the times where we had no exceptions for all the programs where the bug was detected at the right place (true positive). Moreover, we included the execution times of the programs where the code was successfully parsed but no bug was found (no bug was triggered at all). We did not sum up the execution times for the test cases where we had exceptions and no bug was triggered.

TABLE IV: Types of exceptions encountered

Exception ID	Error types
101	Jump node, C goto statement
102	java.lang.ClassCastException: org.eclipse.cdt.internal.core.dom.parser .cpp.CPPFunction cannot be cast to org.eclipse.cdt.internal .core.dom.parser.c.CFunction
103	java.lang.ClassCastException: org.eclipse.cdt.internal.core.dom.parser .cpp.CPPTemplateTypeParameter cannot be cast to org.eclipse .cdt.core.dom.ast.IBasicType
104	java.lang.NullPointerException smtcodan.interpreter.FunctionSpaceStack .enterFunction(FunctionSpaceStack.java:463)

Table IV presents four types of exceptions that we have encountered during our experiments. The exceptions appeared because of current limitations of our framework. Note, that the ID numbers [101, 104]—used in table IV first column—were freely chosen to better depict the encountered exceptions. The exception ”ID 101” was caused by the presence of the C language ”goto” statement when present in the source code. Our plug-in checker triggered bugs in almost all programs except the C++ programs and the programs containing the ”goto” statement. ”ID 102” was encountered because our framework can not currently deal with C++ functions, only C functions. ”ID 103” appeared when trying to convert a C++ template type parameter into an IBasicType which were not compatible. The exception, ”ID 104”, was encountered inside the method enterFunction(SymFunctionCall nextCall, SymFctSignature fsign) which resides in FunctionSpaceStack. The reason is that we do not create symbolic function signatures for C++ function declarations. This limitations will be removed by making the CFG builder algorithm aware of object instantiations and other C++ specific language semantics.

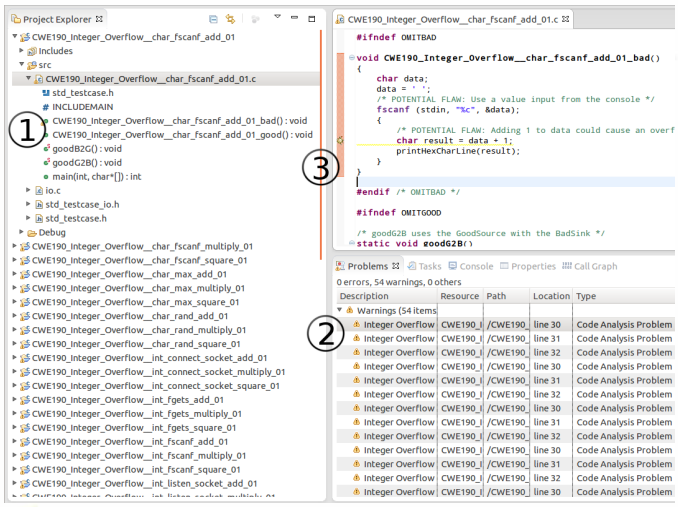

Fig. 3: CWE_190 baseline programs bug reports

Figure 3 presents the bug reports after running our integer overflow checker on the 54 baseline programs (each contains one CFV). The circled numbers in figure 3 represent: number ① depicts the inner structure of the first baseline program, number ② contains 54 bug reports for the 54 true positives contained in the 54 baseline programs (each contains the first CFV) and number ③ indicates the bug location (file name and line number) of the first bug report indicated in the Eclipse ”Problems” view. Furthermore, the user has the possibility to trace back the detected bug—for the programs where bug reports were generated—to the bug location (file name and line number) by double-clicking on one of the bug reports depicted in figure 3 with number ②.

However, by excluding all C++ programs and those programs containing the C language ”goto” statement (no bug reports were generated and contain 11 CFV per baseline test case) a total of 594 (11×54), 22.9% (594 out of 2592) test programs can be excluded meaning that only 1998 (2592 – 594) programs are actually analyzable by our framework. Among 1998 programs containing the same number of bugs (true positives) we successfully detected the bugs in 1908 programs (true positives), which results in a successful coverage of 95.49% with no false positives. In the rest 90 programs (1998 – 1908), 4.51% from the total analyzable programs (1998), our checker did not trigger any true or false positives. Thus, the above results confirm that our approach is accurate and effective in detecting integer overflows.

E. Threats to Validity

Internal Validity: The experimental results presented in table I may not support our findings for several reasons. If our checker incorrectly skips some bugs or misses some constraints, then it might report false positives (we did not encounter any in our experiment). For this reason we carefully extended the StatementProcessor in order to be capable to deal with the new C language semantics present in CWE_190. If we misinterpreted the timing results, then the potential total execution time would not be feasible. We addressed these factors by testing our tool extensively on the open source programs contained in CWE_190 which have a known number of integer overflow bugs. We repeated each run for three times for each of the analyzed programs in order to be sure that the obtained results are right.

External Validity: The results cannot be potentially generalized for C++ programs since the open source programs contained in CWE_190 were not analyzed due to current engine limitations. Our engine does not support C++ language semantics due to the fact that we currently focused on only the C language. We mitigate this issue by arguing that extending the StatementProcessor to be capable to translate every kind of C++ language semantics in SMT-Lib statement is just a matter of time. On the other hand the current CFG builder algorithm is not aware of C++ control flow related semantics (e.g., object instantiation, etc.). The required C++ data types are available in the Codan API and will be implemented into our CFG builder algorithm so that the CFG will contain program execution paths based on specific C++ language control flow semantics.

VI. Discussion and Future Work

We presented an integer overflow checker which runs on C source code and can detect integer overflows with a true positives rate of 95.49% with no false positives. Our checker was developed as an Eclipse plug-in and was used to automatically detect bugs in 2592 programs. We used jUnit tests in order to assess the accuracy of our integer overflow checker automatically. Furthermore, we demonstrated that an integer overflow checker with low false positives rate can be built by formal modeling of C language semantics which are related to integer overflows and with function models which are used to simulate the program environment.

In future we want to extend the CFG building algorithm in order to support C++ control flow related semantics—object instantiation, etc. Additionally, we want to implement other symbolic function models which are relevant for detection of other types of integer related vulnerabilities.

Acknowledgments

This research is funded by the German Ministry for Education and Research (BMBF) under grant number 01IS13020.

References

[1] Aho, A. V. et al. A minimum-distance error-correcting parser for context-free languages. *SIAM Journal of Computation*, 1972.

[2] Ashcraft, K. et al. Using programmer-written compiler extensions to catch security holes. *Proceedings of the IEEE S&P'02*, 2002.

[3] Barrett, C. et al. The SMT-LIB Standard Version 2.0. *Dec.*, 2010.

[4] Barrett, C. et al. Codan- C/C++ static analysis framework for CDT. *In: EclipseCon*, 2011.

[5] Brumley, D. et al. RICH: Automatically protecting against integer-based vulnerabilities. *Proc. of the NDSS'07*, 2007.

[6] Ceesay, E. N. et al. Using type qualifiers to analyze untrusted integers and detecting security flaws in C programs. *Proc. of the DIMVA'06*, 2006.

[7] Chen, P. et al. Brick: A binary tool for run-time detecting and locating integer-based vulnerability. *Proc. of the ARES'09*, 2009.

[8] Chinchani, C. et al. ARCHERR: Runtime environment driven program safety. *Proc. of the ESORICS'04*, 2004.

[9] Clarke, E. M. et al. A tool for checking ANSI-C programs. *Proc. of the TACAS'04*, 2004.

[10] Coker, Z. et al. Program transformations to fix C integers. *Proc. of the ICSE'13*, 2013.

[11] Dietz, W. et al. Understanding integer overflow in C/C++. *Proc. of the ICSE'12*, 2012.

[12] de Moura, L. and Bjørner, N. Z3: an efficient SMT solver. *Proc. of the TACAS'08/ETAPS'08*, 2008.

[13] Gamma, E. et al. Design Patterns. Elements of Reusable Object-Oriented Software, *Addison-Wesley*, 1994.

[14] Godefroid, P. et al. Automated whitebox fuzz testing. *Proc. of the NDSS*, 2008.

[15] Ibing, A. Path-Sensitive Race Detection with Partial Order Reduced Symbolic Execution. *Proc. of the WSFMDS'14*, 2014.

[16] Ibing, A. et al. A Fixed-Point Algorithm for Automated Static Detection of Infinite Loops. *16th International Symposium on High Assurance Systems Engineering, Proc. of the HASE*, 2015.

[17] Ibing, A. Symbolic Execution Based Automated Static Bug Detection for Eclipse CDT, *International Journal on Advances in Security*, 2015.

[18] Ibing, A. Parallel SMT-constrained symbolic execution for Eclipse CDT/Codan. *Int. Conf. Testing Software and Systems (ICTSS)*, 2013.

[19] Khedker, U. et al. Data Flow Analysis. *CRC Press*, 2009.

[20] Le Berre, D. and Parrain, A. The SAT4J library, release 2.2, system description. *JSAT 7*, 2010.

[21] Microsoft PREfast analysis tool. *Microsoft Corporation*.

[22] Mitre Corporation 2011 CWE/SANS Top 25 Most Dangerous Software Errors. *https://cwe.mitre.org/top25/*, 2011.

[23] Mitre Corporation CVE-2002-0639: Integer overflow in sshd in OpenSSH. *http://cve.mitre.org/cgi- bin/cvename.cgi?name=CVE-2002-0639*, 2002.

[24] Mitre Corporation CVE-2010-2753: Integer overflow in Mozilla Firefox, Thunderbird and SeaMonkey. *http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2753*, 2010.

[25] Mitre Corporation CVE-2002-1490: Integer overflow in NetBSD 1.4. *https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1490*, 2002.

[26] Molnar, X. L. D. et al. Dynamic test generation to find integer bugs in x86 binary Linux programs. *Proc. of the 18th USENIX Security Symposium*, 2009.

[27] Moy, Y. et al. Modular Bug-finding for Integer Overflows in the Large: Sound, Efficient, Bit-precise Static Analysis. *MSR-TR-2009-57*, 2009.

[28] Nethercote, N. and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *Proc. of the PLDI'07, ACM*, 2007.

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

[30] Sarkar, D. et al. Flow-insensitive static analysis for detecting integer anomalies in programs. *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference, ACTA Press*, 2007.

[31] Sharir, M. and Pnueli, A. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, 1981.

[32] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages*, 1995.

[33] NIST, Juliet Test Suite v1.2 for C/C++, *online: http://samate.nist.gov/SARD/testsuites/juliet/Juliet_Test_Suite_v1.2_for_C_Cpp.zip*

[34] Wang, T. et al. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. *Proc. of the NDSS'09*, 2009.

[35] Wojtczuk, R. UQBTng: A tool capable of automatically finding integer overflows in Win32 binaries. *Chaos Communication Congress*, 2005.