Architecture Description Language Based Retargetable Symbolic Execution

Andreas Ibing
Chair for IT Security
TU München
Boltzmannstrasse 3, 85748 Garching, Germany

***Abstract*—This paper presents an approach to retargetable SMT-constrained symbolic execution of machine code. The retargetability is based on an existing open-source processor architecture description language which has been used for processor design and automatic generation of toolchains for dynamic program analysis. The benefit of the presented approach is that with a given architecture description, no manual writing of an instruction set grammar or of a translation of instruction semantics into logics is necessary. The proposed tool architecture relies on language reflection, code generation and dynamic loading to retarget symbolic execution to different machine code syntax. Instruction semantics is translated into SMT bit-vector logic equations by symbolically interpreting the architecture description language. The approach is implemented as plug-in extension to the Eclipse IDE and evaluated by automatically detecting integer overflows in binaries for the ARMv5 and SPARCv8 architectures.**

I. Introduction

Symbolic execution [1] is a static program analysis method in which 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. Symbolic execution has been used to automatically analyze source-code, intermediate code and binaries (machine code). A detailed overview of the current state and available tools is given in [2], [3].

Symbolic execution of binaries is applied for the detection of certain bug classes and for the quantification of worst-case execution time (WCET) and worst-case memory consumption (WCMC). One motivation is that certain bugs like integer overflows are processor dependent. Programmers often make assumptions about the underlying data model, e.g. that 'pointer' and 'long' have equal length, or that 'int' and 'long' have equal length. Such assumptions lead to errors when changing the target architecture [4]. Another motivation is that analysis on the source-code level does not prevent from compiler errors. Further, the analysis of WCET and WCMC needs knowledge about the hardware, like instruction latencies and memory hierarchy. Symbolic execution of binaries is described e.g. in [5]. An overview of current methods for determination of WCET is given in [6]. Processor cycle accurate symbolic execution is used for WCET determination in [7], [8].

Architecture description languages (ADL) have been developed for virtual prototyping of new processors. Changes of the processor architecture cause changes in software execution properties like WCET and architecture dependent software bugs. To dynamically analyze software on an application-specific instruction set processor (ASIP) during design space exploration in hardware/software co-design, the software must be compiled and assembled with a toolchain which is retargeted to the instruction set of interest. Such automatic retargeting of the software toolchain comprising compiler, assembler, simulator, debugger and binary utilities is supported by current ADLs. The usage of ADLs for automatic design space exploration is described in [9]. Optimization of the instruction set of a processor for certain applications (normally by adding some special instructions), resulting in an ASIP, is described e.g. in [10]. A current open-source ADL is ArchC [11], which is based on the standardized and widely used SystemC hardware description language. Toolchain generation based on an ArchC model is described in [12], [13]. An ArchC architecture description includes registers, the instruction set including coprocessor instructions, flags, endianness, exceptions and processor modes and the application binary interface (ABI, calling conventions etc.). It further includes the instruction semantics using SystemC notation. There are several processor models already available in ArchC, e.g. ARMv5, SPARCv8, MIPS-I, PowerPC [14].

This paper presents an approach to retargetable symbolic execution. The motivation is that previous approaches require porting of symbolic execution to support a new processor, i.e. manual specification of the grammar of a binary (syntax) and the translation of instructions into logic formulas (semantics). New processors are normally specified in an ADL anyway to generate compiler, simulator and debugger. The proposed approach takes advantage of the available architecture description to automatically retarget symbolic execution and thus to get rid of the porting effort. This enables static analysis and the detection of errors which only occur on the target hardware, even if the target hardware is co-designed at the same moment. A proof-of-concept implementation is presented as plug-in extension for Eclipse CDT.

The remainder of this paper is organized as follows: Section II gives a high-level overview of the tool flow with retargetable symbolic execution. Section III explains the chosen tool achitecture and integration into the Eclipse IDE. The implementation is depicted in Section IV. Then the symbolic interpreter is described, which translates arbitrary instruction set semantics (written in ADL) into logic equations and interprets them with the help of an SMT-solver. In Section V the approach is used to automatically find integer overflows in software compiled to ARM and SPARC architectures. Related work is considered in Section VI. Section VII discusses results and further applications.

978-3-9815370-4-8/DATE15/©2015 EDAA

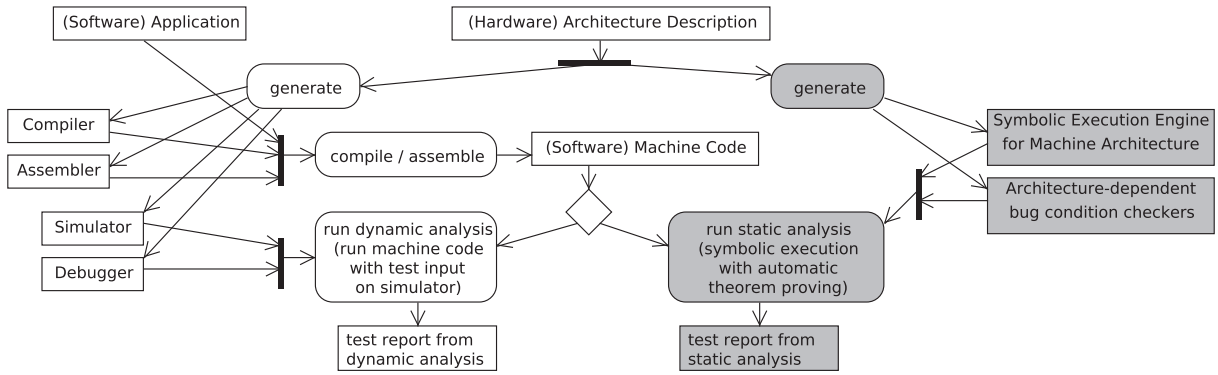

Fig. 1. Tool flow, the proposed approach is shaded.

II. High-Level Tool Flow Description

The tool flow is illustrated as activity diagram in Figure 1. Hardware/software co-design starts with a hardware architecture description and software source code. The left side of the figure shows the standard approach with automatic generation of the binary toolchain and simulator from the architecture description. The system is evaluated by compiling the software with the generated tool chain and running the software with test case input on the generated simulator (dynamic analysis). The proposed approach uses retargetable static analysis with symbolic execution in addition to the retargetable dynamic analysis. The approach is illustrated shaded on the right side of Figure 1. Symbolic execution can automatically find many common software weaknesses like integer overflows or buffer overflows (which might be caused for example by missing input validation), and symbolic execution does not require any (hand-written) test cases. Such errors are not necessarily found by dynamic analysis: they can be missed even when a high code coverage criterion like modified condition/decision coverage (MC/DC) is achieved with test cases, simply when the respective test case is missing together with the code lines. As noted in the introduction, symbolic execution can further be used to automatically quantify WCET and WCMC, which are difficult to quantify with dynamic analysis, because the corresponding program input to trigger the worst-case situation is unknown. While dynamic analysis needs a big enough test suite to achieve sufficient code coverage, the code coverage of symbolic execution is in principle only limited by the available computational power. Static analysis is applied in addition to dynamic analysis, not as replacement. Finding incorrect functionality (logic bugs, as opposed to the mentioned common weaknesses) still relies on dynamic testing with a test suite. The right side of Figure 1 shows that the symbolic execution engine is retargeted based on the architecture description. The retargeting comprises both syntax and semantics of the architecture's instruction set. Software machine code is obtained in the same way as for dynamic analysis, by compilation and assembly with the generated tool chain (a different compiler can also be used if available). In order to automatically check architecture dependent bug conditions, a possibility to explicitly state a bug condition in the architecture description is proposed. As with retargetable dynamic analysis, the system can be automatically re-evalauted for a changing hardware architecture description and/or changing application software.

III. Tool Architecture and IDE Integration

The hardware architecture description specifies the processor and the memory hierarchy. It also specifies the instructions, which comprises both syntax (bit fields) and semantics for each instruction. The instruction semantics is specified in a language which can be parsed into an abstract syntax tree (AST), from which control flow graphs (CFGs) can be generated. A processor iteratively loads and then executes an instruction from an address given in a special register, which is the instruction pointer register. For each instruction, there is a corresponding CFG specifying its semantics.

For symbolic execution of a binary, first the syntax of the instruction set of the target processor must be known in order to parse the instructions. Then, the instruction semantics must be known in order to translate instructions into logic equations for the solver. In the proposed architecture for retargetable symbolic execution, the machine code parser is retargeted with code generation using a parser generator. The machine code symbolic interpreter is not retargeted with code generation, but by directly interpreting the language in which the instruction semantics is specified. Symbolic execution then results in nested interpretation on two levels. On the outer level, machine instructions are interpreted using the register-based interpreter pattern. On the inner level (i.e. for each instruction), the instruction handling semantics methods (CFGs) are interpreted using the tree-based interpreter pattern. Symbolic variables are stored and retrieved during interpretation with a memory space model.

As concrete case, the ArchC ADL is considered and static analysis is integrated into the Eclipse IDE. ArchC relies on SystemC to specify the instruction semantics, which is a C++ class library. As an advantage, Eclipse CDT's code analysis framework can be used, which offers a C/C++ parser, abstract syntax tree visitor and control flow graph builder. As parser generator, ANTLR is used. Moreover, Java features dynamic class loading and a reflection API, which is convenient for retargeting and reloading a machine code parser class with code generation. The reflection API is used to resolve instruction handling method names, which are specified within the architecture description.

The logic backend for deciding branch satisfiability and the presence of bugs is an SMT solver, which is configured to use the SMTlib [15] sublogic of closed quantifier-free formulas with arrays, uninterpreted functions and bitvectors (QF_AUFBV). In CDT's control flow graph nodes, each node

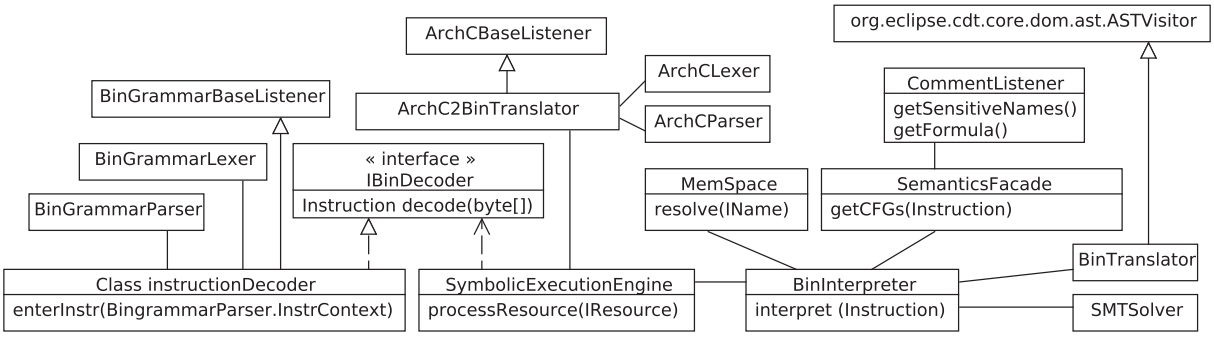

Fig. 2. Architecture overview.

has a reference to the corresponding subtree of the source file's abstract syntax tree (AST). The translation of an instruction's semantics from C++ into SMTlib logic equations can therefore be conveniently implemented with the visitor pattern.

To support the specification of architecture dependent bug conditions in the architecture description, there were two possibilities. The first was to extend ArchC, the second was to add the specification in comments and to use a corresponding comment parser (in the sense of extended static checking). In order to work with the original ArchC grammar, the second option was chosen. To generate the comment parser, again the ANTLR parser generator is used.

The chosen architecture is depicted in Figure 2. The main classes and their functionality are:

- `SymbolicExecutionEngine`: controls retargeting and symbolic execution. It is called from an extension of the Eclipse GUI.
- `ArchC2BinTranslator`: generates a machine code grammar from an architecture description (described in detail in Subsection IV-B1). The ANTLR parser generator is used to generate the retargeted machine code parser (`BinGrammarLexer` and `BinGrammarParser` classes) from the generated grammar.
- `InstructionDecoder`: decodes instructions from a binary program and resolves their names and parameters into those of the architecture description (described in detail in Subsection IV-B2).
- `SemanticsFacade`: resolves CFGs for SystemC instruction semantics methods.
- `BinInterpreter`: gets CFGs for instruction semantics from `SemanticsFacade`, tracks symbols with `MemSpace` and lets `BinTranslator` translate CFG nodes into SMT logic; calls SMT solver to check satisfiability of CFG branch nodes.
- `MemSpace`: used to store and resolve symbolic values (logic formulas) for variables during symbolic interpretation.
- `BinTranslator`: implements Eclipse CDT's `ASTVisitor`, translates instruction semantics from CFG nodes of SystemC methods into SMT logic.
- `SMTSolver`: wraps the solver. Solves satisfiability queries for branches in CFG and for bug conditions.
- `CommentListener`: parses architecture dependent bug condition specifications, resolves contained AST names and generates bug condition formulas.

IV. Implementation

A. Review of ArchC Architecture Description Language

ArchC contains C++ template classes for register bank, pipeline stages, storage (e.g. cache, RAM) and others. Registers, assembler syntax and machine code syntax are described in an ArchC file *arch*_isa.ac. One register is denoted as program counter register (keyword `ac_pc`). Instruction syntax is described hierarchically with instruction format, instruction and format fields (constants and parameters). The instruction semantics is specified in the SystemC file *arch*_isa.cpp, as member functions of the architecture class. Corresponding to the hierarchical syntax description, the semantics of each instruction comprises three handling methods. There is a general instruction handling method for all instructions, e.g. to increment the program counter register. Then there is a method for instruction format specific operations, like shifting an immediate operand. The third method is an instruction specific handling method. The `acsim` program generates a simulator from the architecture description, the `acbingen` program generates binary unitilities.

B. Retargeting Machine Code Parser and AST Listener to Instruction Set

This subsection describes the generation of grammar, parser and AST listener for machine code from the ADL.

1) Translate Architecture Description to Machine Code Grammar: For translation of an architecture description into a grammar of the binary format, the class `ArchC2BinTranslator` was written (compare Figure 2). To parse ArchC, first an ArchC grammar was written in ANTLR syntax. The ANTLR parser generator was used to automatically generate ArchC lexer, parser and listener from this grammar. `ArchC2BinTranslator` extends the generated `ArchCBaseListener`. The input to the translation is illustrated in Figure 3 with a small part of the parse tree of the ARMv5 architecture description. The figure illustrates the mapping of instructions to the corresponding instruction formats. For any ArchC input, `ArchC2BinTranslator` generates the corresponding grammar of the machine code, in ANTLR syntax. This grammar specifies bits on the lowest level, then format fields, then instructions with parameters. The grammar consists of architecture independent header and lexer rules, the in-between depends on the architecture. A machine code parser's parse tree for an instruction in an actual machine

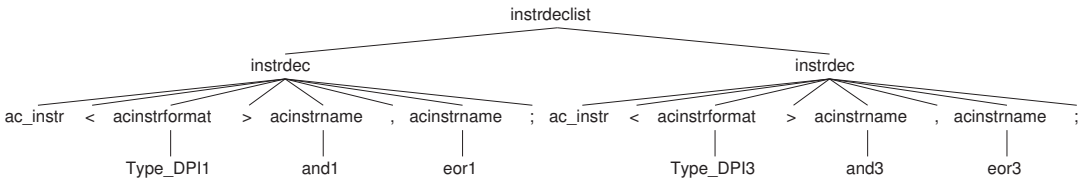

Fig. 3. Part of the parse tree of an architecture description (ARMv5 model) specifying the mapping of instructions to instruction formats.

code example is shown in Figure 4, which also illustrates the hierarchy of general instruction, instruction format, specific instruction, bit fields and bits.

2) Generate Machine Code Parser and AST Listener: To offer access to a binary program in a form convenient for symbolic execution, the class `InstructionDecoder` has been written (compare Figure 2). ANTLR is run on the generated grammar of the binary, to generate lexer, parser and AST listener for the binary format. Node types in the AST of a binary are a-priori unknown to `InstructionDecoder` on several levels (apart from the lexer bit level at the very bottom and a general instruction level above the processor differences). `InstructionDecoder` extends the generated `BingrammarBaseListener` and listens only on exit from AST nodes of the architecture independent instruction level, which is defined in the standard header of the grammar of binaries. `InstructionDecoder` uses the Java reflection API to resolve names of instruction handling methods and format field parameters from the architecture description, which are found as children of an instruction node in the AST. The generated classes are compiled with the `JavaCompiler` and dynamically loaded with the Eclipse plug-in's `ClassLoader`. Also `InstructionDecoder` is dynamically loaded before symbolic execution, and its `decode()` method is accessed through the interface `IBinDecoder`. Figure 4 shows part of a parse tree for an ARMv5 binary, which has been parsed with a generated and dynamically loaded machine code parser.

C. Two-Layer Symbolic Interpretation

This subsection describes the functionality of the classes `BinInterpreter` and `BinTranslator` from Figure 2. The interpretation uses some a-priori knowledge about the ADL, but no a-priori knowledge about any concrete architecture.

1) Outer layer — Register-based interpretation: The interpretation is based on the registers given by the architecture description. The program counter (`ac_pc`) gives the program address of the next instruction to interpret, either as concrete number or as nonambiguous equation system of logic formulas. The start address is given in the ELF header of the binary file. For each instruction, the handling methods are then interpreted on the inner layer. The handling methods are resolved with their names from the architecture description and called with the corresponding parameters (instruction fields with names and values, and register bank object). For conditional branch instructions both branches can be satisfiable. A branch is explicit in a control flow graph on the inner layer.

2) Inner layer — Tree-based interpretation: Corresponding to the syntax of the instruction set, there are normally three semantics methods per instruction. One general method for all instructions, one for the specific instruction format and one for the specific instruction. First, this description (*arch*`_isa.cpp`) is parsed with CDT's already available C/C++ parser to generate an AST. Then CDT's `ControlFlowGraphBuilder` is used to generate a control flow graph from each method's AST (AST subtree rooted in a function definition). The methods' CFGs are cached and resolved by the class `SemanticsFacade` (compare Figure 2). The translation into SMTlib logic works per CFG node and is based on [16]. For each CFG branch node, the current path is validated with a solver satisfiability query. In case of multiple satisfiable branches for one decision node, branch decision backtracking is used as in [16], i.e., first one of the two possible paths is analyzed, the other one later. Figure 5 illustrates the CFG corresponding to the ARMv5 `add` instruction.

3) Translation into Logic: The translation uses the SMTlib sublogic of closed quantifier-free formulas with arrays, uninterpreted functions and bitvectors (QF_AUFBV). A node in the instruction handling method's CFG references a subtree of the method's AST. The class `BinTranslator` extends CDT's `ASTVisitor` class to translate this subtree into SMTlib logic. The translation is performed with a bottom-up traversal of the subtree with the visitor pattern (in the visitor's `leave()` methods). AST names and corresponding values (logic formulas) are stored and resolved with the `MemSpace` class (Figure 2). Single assignments are used for symbolic variables to avoid destructive updates. Variable values are formulas which can depend on arbitrary program input. This enables satisfiability checks for the presence of bugs for any program input. Operations in SystemC are essentially bitvector operations, which are mapped to SMTlib operations on bitvectors. Bitvector lengths are given by the SystemC data types. The equation system corresponding to a path constraint can be queried from the `MemSpace` object. For a CFG branch node, a corresponding boolean valued symbolic variable is generated and path satisfiability is checked with the `SMTSolver` class. As logic backend the Z3 solver is used [17]. An example for a CFG branch is shown in Figure 5, which illustrates the CFG for the ARMv5 `add` instruction. The solver is also called to resolve the concrete value of the next instruction's address from a nonambiguous equation system, using the solver's model generation functionality with the SMTlib `get-value` command.

V. Experiments

The retargetable symbolic execution is tested by automatically checking binaries for different architectures for the presence of integer overflow bugs. The available architecture models for ARMv5 and SPARCv8 are used as examples. The integer overflow bug condition is not explicit in the instruction semantics. This is illustrated by Figure 5, which shows the

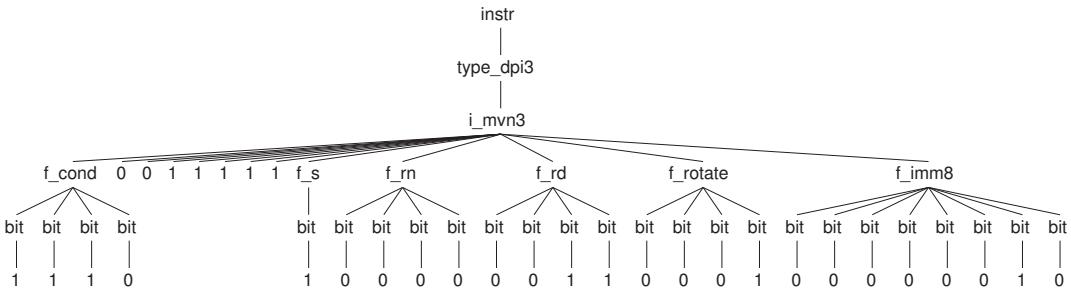

Fig. 4. Part of the parse tree of an ARMv5 binary (corresponding to the instruction: `mvn r3, #-2147483648` from Listing 2).

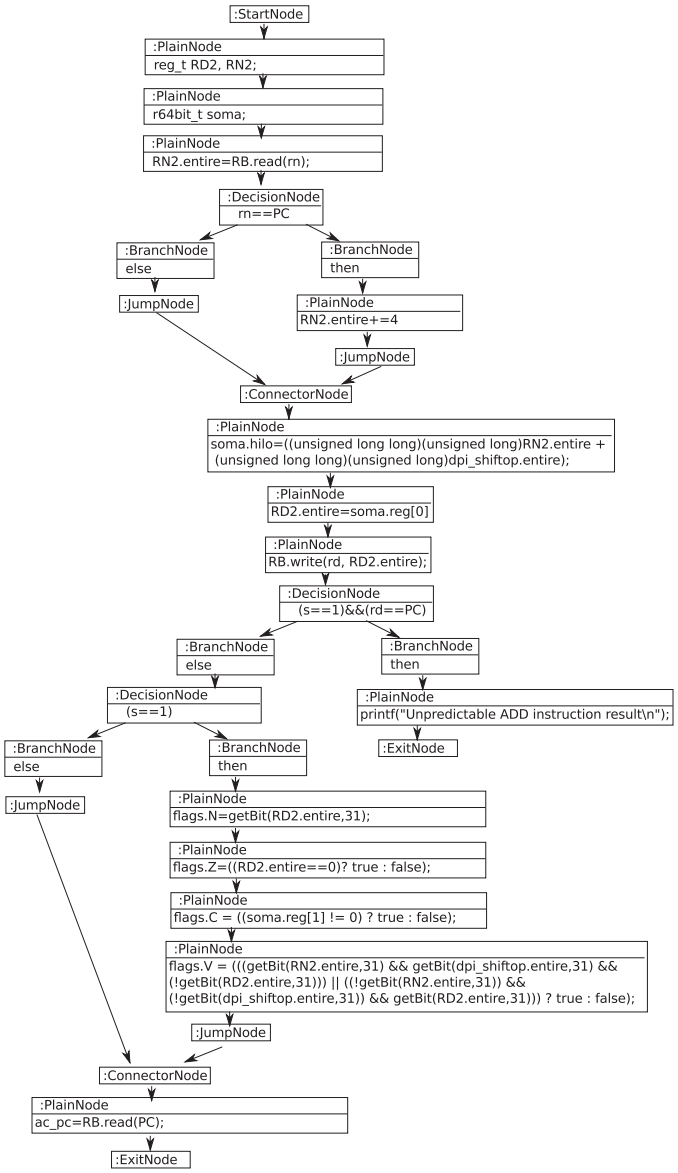

Fig. 5. Control flow graph for the semantics of the ARM `add` instruction.

semantics of the ARMv5 `add` instruction. The computation is specified with 64 bit, in the plain node `'soma.hilo=...'` with type unsigned long long. However, the result which is written to the register bank are only the lower 32bit (`'soma.reg[0]'`). The bug condition can be made explicit by adding the following comment line in the ARMv5 model `armv5e_isa.cpp`:

$$//\texttt{ac\_overflow} : (\texttt{flags.V} == \texttt{true})$$

In the SPARCv8 model `sparc_isa.cpp` similarly the following line can be added:

$$//\texttt{ac\_overflow} : (\texttt{PSR\_icc\_v} == \texttt{true})$$

Checking whether it is satisfiable that a processor's overflow flag evaluates to one is adequate when the processor features such a flag and when the test code always updates it. On some architectures the code may chose not to update the processor's status registers. In that case the overflow condition is a little longer and corresponds to the formula which is used in the instruction semantics to update the overflow flag (compare Figure 5 for the `flags.V` update on ARM). The architecture-dependent bug conditions are parsed by the `CommentParser` (compare Figure 2) and automatically checked during retargeted symbolic execution. The AST names included in the formula are resolved before symbolic execution. The keyword `ac_overflow` indicates the overflow condition and marks the check location. If in global scope, the corresponding symbolic variable(s) are marked with a write trigger. When there is an assignment to one of these variables (in this case when the processor flag is written), a callback to `BinInterpreter` is trigggered and the overflow condition is checked with the solver. As test case, an example from the Juliet test suite for static analyzers [18] is used, namely `CWE190_Integer_Overflow__int_max_add_01`.
The Juliet test cases contain 'good' functions as well as 'bad' functions to adequately determine a checker's accuracy in terms of false negative and false positive bug detections. The test case was simplified by removing calls to the standard C library. This is because the C library is linked dynamically, and calls to dynamic libraries are currently not supported. The 'bad' C function is shown in Listing 1. The program is compiled for ARMv5 and SPARCv8. Parts of the binaries (disassembled with retargeted `objdump`) containing the overflow are illustrated in Listing 2 for ARM, and in Listing 3 for SPARC. The overflows are successfully and automatically detected in the binaries for both architectures.

VI. Related Work

[19] and [20] are model checkers for SystemC (hardware) designs. They do not feature symbolic execution of software running on the specified hardware. Special-purpose description languages have been developed for bug finding in binaries on the one hand, or for WCET analysis on the other. [21] is a binary analysis framework for security flaws. A major difference to the approach presented here is that it transforms binaries to an intermediate presentation before analysis. So to support a new processor, only the new frontend has to be specified. Bug detection in binaries based on a 'transformer specification language' is presented in [22]. This transformer specification language is not a full-featured ADL. To support a new processor, the corresponding transformer description must be written. [21] and [22] also do not feature cycle-accurate symbolic execution (e.g. for WCET analysis). Another description language, this time targeted towards static WCET analysis, is described in [23]. To determine WCET on a new processor, the processor's pipeline must be specified. Also [23] is not a full-featured ADL and does not support generation of a binary toolchain.

Listing 1. C function from Juliet test case.

```
void CWE190_Integer_Overflow_int_max_add_01_bad()
{
    int data;
    ...
    data = INT_MAX;
    ...
    /* POTENTIAL FLAW */
    int result = data + 1;
    printIntLine(result);
}
```

Listing 2. ARM instructions.

```
118: e3a03000    mov   r3, #0       ; 0x0
11c: e50b3010    str   r3, [fp, #-16]
120: e3e03102    mvn   r3, #-2147483648
124: e50b3010    str   r3, [fp, #-16]
128: e51b3010    ldr   r3, [fp, #-16]
12c: e2933001    add   r3, r3, #1  ; 0x1
```

Listing 3. SPARC instructions.

```
108: 03 1f ff ff   sethi  %hi(0x7ffffc00), %g1
10c: 82 10 63 ff   or  %g1, 0x3ff, %g1
110: c2 27 bf f4   st  %g1, [ %fp + -12 ]
114: c2 07 bf f4   ld  [ %fp + -12 ], %g1
118: 82 80 60 01   inccc  %g1
```

VII. Discussion

The benefit of the presented approach is that with a given architecture description, no manual writing of an instruction set grammar or of a translation of instruction semantics into logics is necessary. Also there is no testing effort of portings to a new processor, as the generation is 'correct by construction'. An architecture description is normally written anyway to generate a toolchain, so that it comes 'for free'. Future work includes the retargetable detection of other bug types. A further application is the retargetable determination of worst-case execution time, e.g. for automatic ASIP optimization with iterative changes to the instruction set. This would be enabled by evaluating the instruction latency description (which ArchC already includes) during retargetable symbolic execution.

Acknowledgement

This work has been 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, pp. 385–394, 1976.

[2] C. Cadar et. al., "Symbolic execution for software testing in practice – preliminary assessment," in *Int. Conf. Software Eng. (ICSE)*, 2011.

[3] 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.

[4] R. Seacord, *The CERT C secure coding standard*. Addison-Wesley, 2009.

[5] T. Reps et. al., "There's plenty of room at the bottom: Analyzing and verifying machine code," in *Int. Conf. Computer Aided Verification (CAV)*, 2010.

[6] R. Wilhelm et. al., "The worst-case execution time problem overview of methods and survey of tools," *ACM Trans. Embedded Computing Systems*, vol. 7, no. 3, 2008.

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

[8] B. Benhamamouchi, B. Monsuez, and F. Vedrine, "Computing WCET using symbolic execution," in *Int. Workshop Verification and Evaluation of Computer and Communication Systems (VECoS)*, 2008.

[9] P. Mishra, A. Shrivastava, and N. Dutt, "Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable SOCs," *ACM Trans. Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 626–658, 2006.

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

[11] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC architecture description language and tools," *Int. J. Parallel Programming*, vol. 33, no. 5, 2005.

[12] A. Baldassin et. al., "An open-source binary utility generator," *ACM Trans. Design Automation of Electronic Systems*, vol. 13, no. 2, 2008.

[13] R. Auler, P. Centoducatte, and E. Borin, "ACCGen: An automatic ArchC compiler generator," in *Int. Symp. Computer Architecture and High Performance Computing*, 2012.

[14] ArchC, "Architecture Description Language," online www.archc.org, visited on 01.09.2014.

[15] C. Barrett, A. Stump, and C. Tinelli, *The SMT-LIB Standard Version 2.0*, Dec. 2010. [Online]. Available: http://goedel.cs.uiowa.edu/smtlib/papers/smt-lib-reference-v2.0-r10.12.21.pdf

[16] A. Ibing, "Parallel SMT-constrained symbolic execution for Eclipse CDT/Codan," in *Int. Conf. Testing Software and Systems*, 2013.

[17] L. de Moura and N. Bjorner, "Z3: An efficient SMT solver," in *Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[18] T. Boland and P. Black, "Juliet 1.1 C/C++ and Java test suite," *IEEE Computer*, vol. 45, no. 10, 2012.

[19] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "Kratos: A software model checker for SystemC," in *Int. Conf. Computer Aided Verification (CAV)*, 2011.

[20] C. Chou, Y. Ho, C. Hsieh, and C. Huang, "Symbolic model checking on SystemC designs," in *Design Automation Conf.*, 2012.

[21] D. Song et. al., "Bitblaze: A new approach to computer security via binary analysis," in *Int. Conf. Information Systems Security*, 2008.

[22] J. Lim and T. Reps, "TSL: A system for generating abstract interpreters and its application to machine-code analysis," *ACM Trans. Programming Languages and Systems*, vol. 35, no. 1, 2013.

[23] S. Thesing, "Safe and precise WCET determination by abstract interpretation of pipeline models," Ph.D. dissertation, Universitaet des Saarlandes, 2004.