SMT-Based Refutation of Spurious Bug Reports in the Clang Static Analyzer

Mikhail R. Gadelha*, Enrico Steffinlongo†, Lucas C. Cordeiro‡, Bernd Fischer§, and Denis A. Nicole†
*SIDIA Instituto de Ciência e Tecnologia, Brazil. E-mail: m.gadelha@samsung.com
†University of Southampton, UK. ‡University of Manchester, UK. §Stellenbosch University, South Africa.

Abstract—**We describe and evaluate a bug refutation extension for the Clang Static Analyzer (CSA) that addresses the limitations of the existing built-in constraint solver. In particular, we complement CSA's existing heuristics that remove spurious bug reports. We encode the path constraints produced by CSA as Satisfiability Modulo Theories (SMT) problems, use SMT solvers to precisely check them for satisfiability, and remove bug reports whose associated path constraints are unsatisfiable. Our refutation extension refutes spurious bug reports in 8 out of 12 widely used open-source applications; on average, it refutes ca. 7% of all bug reports, and never refutes any true bug report. It incurs only negligible performance overheads, and on average adds 1.2% to the runtime of the full Clang/LLVM toolchain. A demonstration is available at** `https://www.youtube.com/watch?v=ylW5iRYNsGA`.

I. Introduction

LLVM comprises a set of reusable components for program compilation [1], unlike other popular compilers, e.g., GCC and its monolithic architecture [2]. Clang [3] is an LLVM component that implements a frontend for C, C++, ObjectiveC and their various extensions. Clang and LLVM are used as the main compiler technology in several closed- and open-source ecosystems, including MacOS and OpenBSD [4].

The Clang Static Analyzer (CSA) [5] is an open-source project built on top of Clang that can perform context-sensitive interprocedural analysis for programs written in the languages supported by Clang. CSA symbolically executes the program, collects constraints, and reasons about bug reachability using a built-in constraint solver. It was designed to be fast, so that it can provide results for common mistakes (e.g., division by zero or null pointer dereference) even in complex programs. However, its speed comes at the expense of precision, and it cannot handle some arithmetic (e.g., remainder) and bitwise operations. In such cases, CSA can explore execution paths along which constraints do not hold, which can lead to incorrect results being reported.

```
1 unsigned int func(unsigned int a) {
2   unsigned int *z = 0;
3   if ((a & 1) && ((a & 1) ^ 1))
4     return *z;
5   return 0;
6 }
```

Fig. 1: A small C safe program. The dereference in line 4 is unreachable because the guard in line 3 is always false.

Consider the program in Fig. 1. This program is safe, i.e., the unsafe pointer dereference in line 4 is unreachable because the guard in line 3 is not satisfiable; `a & 1` holds if the last bit in `a` is one, and `(a & 1) ^ 1` inverts the last bit in `a`. The analyzer, however, produces the following (spurious) bug report when analyzing the program:

```
main.c:4:12: warning: Dereference of null
 pointer (loaded from variable 'z')
    return *z;
           ^~
1 warning generated.
```

The null pointer dereference reported here means that CSA claims to nevertheless have found a path where the dereference of `z` is reachable.

Such spurious bug reports are in practice common; in our experience, about 50% of the reports in large systems (e.g., git) are actually spurious. Junker et al. [6] report similar numbers for a similar static analysis technology. Identifying spurious bug reports and refactoring the code to suppress them puts a large burden on developers and runs the risk of introducing actual bugs; these issues negate the purpose of a lightweight, fast static analysis technology.

Here we present a solution to this conundrum. We first use the fast but imprecise built-in solver to analyze the program and find potential bugs, then use slower but precise SMT solvers to refute (or validate) them; a bug is only reported if the SMT solver confirms that the bug is reachable. We implemented this approach inside the CSA and evaluated it over twelve widely used C/C++ open-source projects of various size using five different SMT solvers. Our experiments show that our refutation extension can remove false bug reports from 8 out of the 12 analyzed projects; on average, it refuted 11 (or approximately 7% of all) bug reports per project, with a maximum of 51 reports refuted for XNU; it never refuted any true bug report. Its performance overheads are negligible and on average our extension adds only 1.2% to the runtime of the full Clang/LLVM toolchain.

II. The Clang Static Analyzer

CSA performs a context-sensitive interprocedural data-flow analysis via graph reachability [7] and relies on a set of checkers, which implement the logic for detecting specific types of bugs [5], [8]. Each path in a function is explored, which includes taking separate branches and different loop unrollings. Function calls on these paths are inlined whenever possible, so their contexts and paths are visible from the caller's context.

Real-world programs, however, usually depend on external factors, such as user inputs or results from library components,

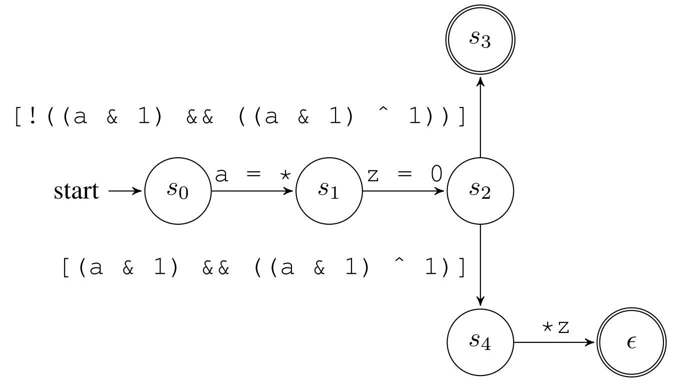

Fig. 2: Exploded graph of the program in Fig. 1, which contains bitwise operations that are not handled by CSA's original constraint solver, generating a spurious bug report.

for which source code is not always available. These unknown values are represented by symbols, and the built-in constraint solver in the static analyzer reasons about reachability based on expressions containing these symbols.

The CSA relies on a set of checkers that are engineered to find specific types of bugs, ranging from undefined behaviour to security property violations, e.g., incorrect usage of insecure functions like `strcpy` and `strcat` [10]. The checkers subscribe to events (i.e., specific operations that occur during symbolic execution) that are relevant to their bug targets; for example, the nullability checker subscribes to pointer dereferences. They then check the constraints in the current path and throw warnings if they consider a bug to be reachable.

These checkers can report incorrect results since the symbolic analysis is incomplete (i.e., they can miss some true bugs) and unsound (i.e., they can generate spurious bug reports). The sources of these incorrect results are approximations in two components, namely the control-flow analysis and the constraint solver.

The control-flow analysis evaluates function calls inside the same translation unit (TU); if the symbolic execution engine finds a function call implemented in another TU, then the call is skipped and pointers and references passed as parameters are invalidated while the function return value is assumed to be unknown. Cross translation unit support (CTU) is under development [8]; it is not part of the CSA main branch yet.

The built-in constraint solver (based on interval arithmetic) was built to be fast rather than precise, and removes expressions from the reasoning if they contain unsupported operations (e.g, remainders and bitwise operations) or are too complex (e.g., contain more than one operator symbol).

The bug reports generated by the checkers are then post-processed before they are reported to the user. In this final step, a number of heuristics are applied to remove incorrect bug reports and to beautify the reports. The reports are also deduplicated, so that different paths that lead to the same bug only generate one report.

As an example of this process, consider the exploded graph [7] in Fig. 2; it represents the graph explored by the analyzer when analyzing the program in Fig. 1. Here, a node s is a pair (V, C), where V is a map $var \rightarrow 2^{128}$ that maps a value to every variable var in the program, and C is a map $var \rightarrow 2^{128} \times 2^{128}$ that maps the interval constraints to every variable var required to reach that node. An edge is a tuple $(s_i, \texttt{OP}, s_{i+1})$, modeling the constraints on a transition from s_i to s_{i+1}, and OP is a operation performed in the transition, either changing a constraint $c \in C$ or a read/write operation over a $var \in V$. We reserve two special symbols: an ϵ node is a property violation and unknown values are shown as "$\star$" symbols.

CSA's "NullDereference" checker adds the transition to ϵ representing a dereference of `z`. Note that no error node is added when the parameter `a` is read (even if it was not explicitly initialized), because parameters are assumed to be unknown rather than of uninitialized values, unless the control-flow analysis infers otherwise. The analyzer will explore all paths in the graph because it cannot infer that the constraint that leads to node s_4 is always false.

III. Refuting False Bugs using SMT Solvers

One approach to address the limitations of the built-in constraint solver is to replace it by an SMT solver. This approach has been implemented in Clang but empirical evaluations show that this approach can be up to 20 times slower.[1]

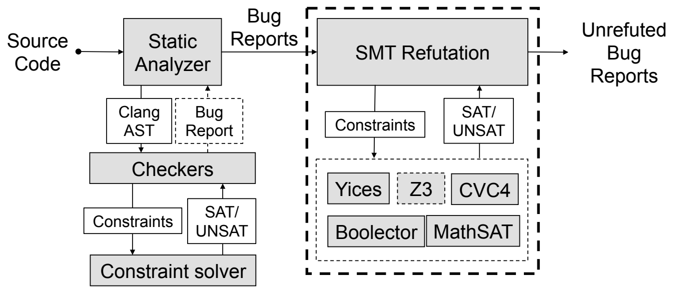

Fig. 3: The refutation extension in the Clang Static Analyzer.

We developed an alternative solution: we use the more precise SMT solvers to reason about bug reachability only in the post processing step. CSA already has heuristics in place to remove incorrect bug reports, so we extended those heuristics to precisely encode the constraints in SMT and to check for satisfiability. Fig. 3 illustrates the architecture of our solution. After the static analyzer generates the bug reports, the SMT-based refutation extension will encode the constraints as SMT formulas and check them for satisfiability. CSA already supports constraint encoding in SMT using Z3 [11] but we also implemented support for Boolector [12], Yices [13], MathSAT [14], and CVC4 [15].

A bug report BR is a straight line graph representing the path to a property violation (i.e., an ϵ-node). Our refutation extension walks backwards through all nodes $s_i \in BR$, collects their constraints, and checks their conjunction for satisfiability. If the formula is unsatisfiable, the bug is spurious.

Our constraint encoding algorithm is shown in Algorithm 1. Let us assume a set of constraints C, an SMT formula Φ, and a method $encode(expr, \Phi)$, which encodes an expression $expr$ in the SMT formula Φ. Algorithm 1 contains two optimizations

[1]https://reviews.llvm.org/D28952

Algorithm 1: $encodeConstraint(cs, \Phi)$

```
Input: A set of constraints $C$ and an SMT formula $\Phi$
Output: The formula $\Phi$ with all constraints $c$ encoded in SMT
1 foreach $c \in C$ do
2     if $c.var \in \Phi$ then continue;
3     if $c.interval.lower == c.interval.upper$ then
4         $encode(c.var == c.interval.lower, \Phi)$
5     else
6         $encode(c.var \geq c.interval.lower \wedge c.var \leq c.interval.upper, \Phi)$
7     end
8 end
```

when encoding constraints: duplicated symbol constraints are ignored (line 2) and if the constraint is a concrete value (the lower bound is equal to the upper bound), the constraint is encoded as an equality (line 3). Note that ignoring symbol constraints in line 2 is only possible because the refutation extension runs from the last node in a bug report (the property violation) to the initial node; any new symbol constraint found when walking backwards will always be a subset of the symbol constraints already encoded.

Fig. 4 shows the SMT formula of the bug found when analyzing the program in Fig. 1. The formula is equivalent to the path $[s_0, s_1, s_2, s_4, \epsilon]$ in Fig. 2 and `$0` is the value of the variable `a`. Since the formula is unsatisfiable, CSA will not produce a bug report for this path.

```
(declare-fun $0 () (_ BitVec 32))
(assert (= ((_ extract 0 0) $0) #b1))
(assert (= ((_ extract 0 0) $0) #b0))
```

Fig. 4: The SMT formula of the bug report from Fig. 1, using Z3. Note that the solver was able to simplify the formula to two assertions: the first bit should be one and zero at the same time. Since this is a contradiction, the formula is UNSAT.

A. Running the Clang Static Analyzer

To run CSA it is enough to use the `scan-build` tool shipped with Clang. The tool runs CSA during the compilation of the project and it is simple to use: instead of running `make`, simply run `scan-build make`. Scan-build offers several options to customize the analysis, including enabling (and disabling) the various checkers in CSA and enabling our refutation extension. Once the build is done, a detailed report is generated for each reported bug.

To analyze a project with our refutation extension enabled, run `scan-build -analyzer-config 'crosscheck-with-z3=true' make`. We developed five different SMT backends in the CSA, however, only Z3 is available as part of the mainstream release of Clang. The backends for Boolector, MathSAT, Yices, and CVC4 are currently only available in https://github.com/mikhailramalho/clang.

IV. Experimental Evaluation

The experimental evaluation of our refutation extension consists of two parts. We first present the research questions, projects evaluated and the environment setup, before we compare the analysis results of the CSA with and without the bug refutation extension enabled; the two approaches are compared in terms of number of refuted bugs and verification time.

A. Experimental Objectives and Setup

Our experimental evaluation aims to answer two research questions:

RQ1 **(soundness)** Is our approach sound and can the refuted bugs be confirmed?

RQ2 **(performance)** Is our approach able to refute spurious bug reports in a reasonable amount of time?

We evaluated the new bug refutation extension in twelve open-source C/C++ projects: *tmux (v2.7)*, a terminal multiplexer; *redis (v4.0.9)*, an in-memory database; *openSSL (v1.1.1-pre6)*, a software library for secure communication; *twin (v0.8.0)*, a windowing environment; *git (v2.17.0)*, a version control system; *postgreSQL (v10.4)*, an object-relational database management system; *SQLite3 (v3230100)*, a relational database management system; *curl (v7.61.0)*, command-line tool for transferring data; *libWebM (v1.0.0.27)*, a WebM container library; *memcached (v1.5.9)*, a general-purpose distributed memory caching system; *xerces-c++ (v3.2.1)*, a validating XML parser; and *XNU (v4570.41.2)*, the operating system kernel used in Apple products.

All experiments were conducted on a desktop computer with an Intel Core i7-2600 running at 3.40GHz and 24GB of RAM. We used Clang v7.0. A time limit of 15s per bug report was set per bug report. All scripts to analyze the projects are available in https://github.com/mikhailramalho/analyzer-projects.

B. Bug Refutation Comparison

Projects	time (s) (no ref)	time (s) (ref)	reported bugs (no ref)	refuted bugs
tmux	86.5	89.9	19	0
redis	347.8	338.3	93	1
openSSL	138.0	128.0	38	2
twin	225.6	216.7	63	1
git	488.7	405.9	70	11
postgreSQL	1167.2	1112.4	196	6
SQLite3	1078.6	1058.4	83	15
curl	79.8	79.9	39	0
libWebM	43.9	44.2	6	0
memcached	96.0	96.2	25	0
xerces-c++	489.8	433.2	81	2
XNU	3441.7	3405.1	557	51
Total	7683.7	7408.5	1270	89

TABLE I: Results of the analysis with and without refutation.

Table I shows the results of CSA with and without bug refutation enabled. Here, *time (s) (no ref)* is the analysis time without refutation, *time (s) (ref)* is the analysis times with refutation enabled, averaged over all supported solvers (Z3, Boolector, MathSAT, Yices and CVC4), *reported bugs (no ref)* is the number of bug reports produced without refutation and *refuted bugs* is the number of refuted bugs. All solvers refuted the same bugs. There were bugs refuted in 8 out of the 12 analyzed projects: redis, openssl, twin, git, postgresql, sqlite3, xerces and XNU. On average, 11 bugs were refuted when analyzing these projects, with up to 51 bugs refuted in XNU.

In total, 89 bugs were refuted and an in-depth analysis of them shows that all of them were false positives, and thus affirm RQ1. Our technique is unable to refute all false bugs as the unsound interprocedural analysis is another source of false positives in CSA, which was not addressed in this work.

The average time to analyze the projects with refuted bugs was 35.0 seconds faster, a 6.25% speed up, and thus affirm RQ2. This is because the static analyzer generates html reports for each bug, which involves intensive use of IO (e.g., the html report produced for the program in Fig. 1 is around 25kB), and by removing these false bugs, fewer reports are generated and the analysis is slightly faster. Out of the four projects, where no bug was refuted (tmux, curl, libWebM and memcached), the analysis was 1.0 second slower on average: a 1.24% slowdown.

We average the analysis time of all solvers because their performance was equivalent. For that reason, we did not submit the other backends to the mainstream Clang; we expected different performance between the solvers but all they refuted the same bugs and the analysis times were within 5% from each other. More solvers offer more flexibility specially license-wise, however, currently their is no demand for having more solvers in the CSA.

V. Related Work

Static analysis of programs has seen a great improvement in the last years and in many cases it has been applied for the analysis of big real-world projects. Frama-C [16] is an extensible analysis framework based on abstract interpretation for the C language. It also has a large number of plugins for checking different program properties. Infer [17] is an open-source static code analysis tool used for the verification of the Facebook code base and on many mobile apps (e.g., WhatsApp) and is adopted by many companies (e.g., Mozilla, Spotify). Cppcheck [18] focuses on detecting dangerous coding practices and undefined behaviours. It was used for detecting many vulnerabilities including a stack overflow in X.org.

The use of SMT solvers as backends for static analysis tools is a well known approach. It has been adopted in the ESBMC [19] C/C++ bounded model checker. It encodes the program as an SMT formula and depending on its satisfiability it detects possible bugs. This encoding technique is proved to be sound, however, it does not scale up to real-world programs.

An approach similar to the one adopted in this paper has also been used in Goanna [6]: a C/C++ static analyzer able to scale up to real-world programs (e.g., OpenSSL, WireShark). After detecting all bugs, false ones are eliminated by analyzing the feasibility of error paths using SMT solvers. The biggest difference between Goanna and our approach is on the first (imprecise) analysis: Goanna uses NuXmv and MathSAT5 to generate the bug reports, while we use a custom built-in constraint solver. Similarly to our approach, they use Z3 to encode and refute false bug reports, but we offer a wider selection of SMT solvers to choose from.

VI. Conclusions

Our SMT-based bug refutation extension in CSA is a simple but powerful extension; it prevents reporting the class of false bugs generated by the unsound constraint solver while introducing minimal overhead to the analysis. Our empirical evaluation shows that the bug refutation extension can consistently reduce the analysis time if bugs are removed, while the slow down when no bug is refuted is negligible. We used five different solvers in our experiments (Z3, Boolector, MathSAT, Yices and CVC4) and their performance is equivalent; our refutation extension using Z3 is already part of the Clang v7.

Our experiments show that the CSA constraint solver is not the largest source of imprecision, as replacing it by a more precise SMT solver leaves an estimated 85% of the spurious bug reports unrefuted. We conjecture that these are caused by the approximations in the TU-based control flow analysis (CFA). We will replace it by a CTU-based CFA during refutation and also extract test inputs from the counterexamples to double-check reachability to increase our precision.

Acknowledgements. We would like to thank George Karpenkov and Artem Dergachev for the guidance during the development of this project and, in particular, George Karpenkov for the initial idea of this project.

References

[1] C. Lattner and V. Adve, “LLVM: A Compilation Framework For Lifelong Program Analysis & Transformation,” *CGO*, pp. 75–96, 2004.
[2] D. Novillo, “GCC- An Architectural Overview, Current Status And Future,” in *The Linux Symposium*, 2006.
[3] C. Lattner, *Clang Documentation*. The Clang-LLVM Project, 2015. [Accessed February-2019].
[4] M. Larabel, “OpenBSD Switches To Clang Compiler For I386/AMD64.” https://www.phoronix.com/scan.php?page=news_item&px=OpenBSD-Default-Clang, 2017. [Accessed February-2019].
[5] M. Arroyo, F. Chiotta, and F. Bavera, “An User Configurable Clang Static Analyzer Taint Checker,” *SCCC*, pp. 1–12, 2016.
[6] M. Junker, R. Huuck, A. Fehnker, and A. Knapp, “SMT-based False Positive Elimination In Static Program Analysis,” *ICFEM*, LNCS 7635, pp. 316–331, 2012.
[7] T. Reps, S. Horwitz, and M. Sagiv, “Precise Interprocedural Dataflow Analysis Via Graph Reachability,” *POPL*, pp. 49–61, ACM, 1995.
[8] G. Horváth, P. Szécsi, Z. Gera, D. Krupp, and N. Pataki, “Implementation And Evaluation Of Cross Translation Unit Symbolic Execution For C Family Languages,” *ICSE*, pp. 428–428, ACM, 2018.
[9] “Available Checkers.” https://clang-analyzer.llvm.org/available_checks.html, 2018. [Accessed February-2019].
[10] L. De Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” *TACAS*, LNCS 4963, pp. 337–340, 2008.
[11] R. Brummayer and A. Biere, “Boolector: An Efficient SMT Solver For Bit-Vectors And Arrays,” *TACAS*, LNCS 5505, pp. 174–177, 2009.
[12] B. Dutertre and L. De Moura, “The Yices SMT Solver,” August 2006.
[13] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, “The MathSAT5 SMT Solver,” *TACAS*, LNCS 7795, pp. 93–107, 2013.
[14] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, “CVC4” *CAV*, LNCS 6806, pp. 171–177, 2011.
[15] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-C: A Software Analysis Perspective,” *FAC*, vol. 27, no. 3, pp. 573–609, 2015.
[16] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O’Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, “Moving Fast With Software Verification,” *NASA Formal Methods*, (Cham), pp. 3–11, Springer International Publishing, 2015.
[17] “Cppcheck - A Tool For Static C/C++ Code Analysis.” http://cppcheck.sourceforge.net, 2018. [Accessed February-2019].
[18] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, “ESBMC 5.0: An Industrial-Strength C Model Checker,” *ASE*, pp. 888–891, 2018.