STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.06

AN INITIAL PROTOTYPE OF TIERED CONSTRAINT SOLVING IN THE CLANG STATIC ANALYZER

RÉKA KOVÁCS AND GÁBOR HORVÁTH

Abstract. Static analysis is a widely used method for finding bugs in large code bases. One of the most popular static analysis tools used for software written in C/C++ languages is the Clang Static Analyzer [1]. During symbolic execution [2] of the source code, the analyzer models path sensitivity by keeping track of constraints on symbolic variables. The built-in constraint manager module, while granting excellent performance, only handles constraints on certain types of integer expressions, which has a detrimental effect on the quality of the analysis, as the infeasibility of certain execution paths cannot be proved. This often leads to false positive findings, i.e. error reports issued for code parts that are actually correct.

This paper records the first milestone in an effort to integrate the state-of-the-art Z3 theorem prover [3] into the Clang Static Analyzer in order to post-process bug reports. While full integration is hindered by the burden Z3 places on the duration of the analysis, the refutation of false positive reports using information collected by the default pass can improve analysis quality substantially while introducing only a moderate regression in performance. We present an initial prototype of the tiered constraint solving solution that is already capable of filtering out some bogus reports, evaluate it on real-world software projects, and explore possible improvements we plan to accomplish in our future work.

1. Introduction

Static analysis is the analysis of software without actually executing programs, usually performed by an automated tool on the source code. Static analysis tools are widely used in the continuous integration chains of production software as their comprehensive checks can provide rapid feedback on the code's performance, reliability, and safety.

Received by the editors: April 2, 2018.

2010 *Mathematics Subject Classification.* 68N20.

1998 *CR Categories and Descriptors.* F.3.2 [**Logics and meanings of programs**]: Semantics of Programming Languages – *Program analysis*.

Key words and phrases. Static analysis, symbolic execution, Clang, SMT solver.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

One of the main considerations behind the design decisions of static analysis tools is the number of false positive reports produced by the tool. False positives are warnings issued incorrectly for code parts that do not contain erroneous behavior. A high ratio of bogus reports is a much greater problem for an industrial bug finding tool than some number of bugs missed (even so as it is impossible to find all bugs using static analysis [5]). Bug reports need to be reviewed by developers one-by-one in order to correct potential errors in their software. If the tool presents an overwhelming amount of false warnings to the developer, its usability suffers and developers lose their trust in the tool.

The Clang Static Analyzer is a symbolic execution engine built atop of clang [4], a relatively recently developed LLVM compiler front-end for C-family languages. The Static Analyzer is an increasingly popular choice of a static analysis tool despite still being a work-in-progress, as it is free, open-source, but its power already matches that of most closed-source tools widespread in the industry.

Despite heavy developer effort, the Static Analyzer suffers from the problem of false positive reports much like other similar tools. One possible way to improve the quality of the reports is to improve the constraint management of symbolic expressions, which plays an important role in proving the infeasibility of impossible execution paths during symbolic execution. An important intermediary step in this direction is the refutation of false positive reports by re-evaluating constraints by a more powerful constraint solver than the one currently built into the engine.

In the following section, we give a brief overview of the inner workings of the analyzer and explain the role of constraint management during the symbolic execution of a program. In Section 3, we present the results of an experiment highlighting the problem with a straightforward solution and explain the motivation behind the choice of the method described in this paper. In Section 4, we explore some aspects of the problem that need to be considered while constructing the solution. Next, we evaluate our prototype on real-world software projects and then raise some questions for future work in Section 6.

2. The Clang Static Analyzer

2.1. **General overview.** The Clang Static Analyzer is a symbolic execution engine capable of analyzing C, C++, and Objective-C programs. Symbolic execution is a form of abstract interpretation of the source code, where each unknown value encountered is assigned a symbol, on which operations are performed symbolically. Along this process, the analyzer attempts to enumerate all possible execution paths by building a so-called *exploded graph* [6]. Each

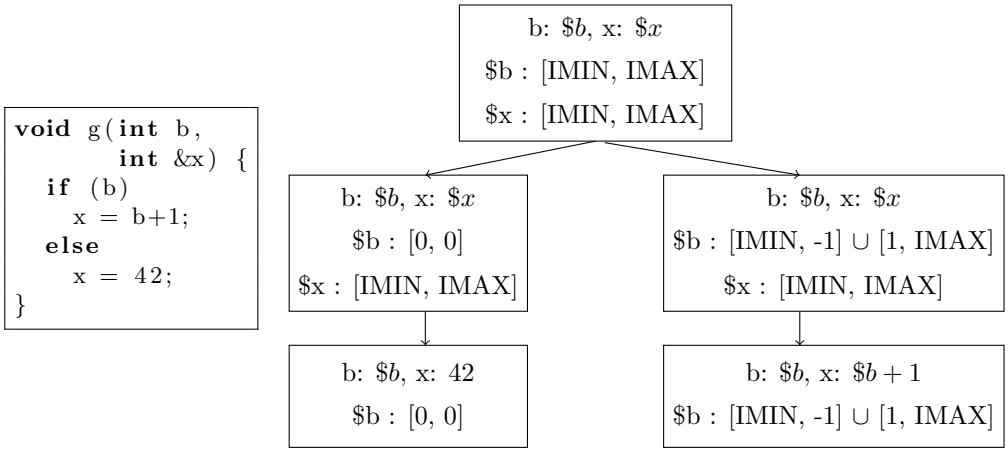

FIGURE 1. An example depicting the representation of symbolic execution in the exploded graph.

vertex of this graph is a *(program state, program point)* pair, where the program point determines the current location in the program, and memory is represented using a hierarchy of memory regions [7]. The program state holds traits of the program such as the *environment*, which records symbolic values of active expressions, and a data structure holding *range constraints*, i.e. ranges that symbolic values may take [8], among others.

During the execution of a path, the analyzer collects constraints on symbolic expressions. The built-in constraint solver module can reason about simpler pointer and integer expressions, by representing constraints on them using closed ranges of integer values. One of the main roles of the constraint manager is to determine whether these constraints become unsatisfiable, in which case the analysis of the current path should be terminated. It is vital for the analyzer to recognize such infeasible program paths in the exploded graph mainly as not to issue error reports for paths that will never be executed.

An example analysis can be seen along with its simplified exploded graph on Figure 1. Function `g` can lead to two execution paths. Since the value of `b` and `x` is initially unknown, these values are represented with symbols `$b` and `$x`, which can take on arbitrary values. As the analysis continues on the execution path corresponding to the `else` branch, the value of `b` is known to be zero, and later we discover that the value of `x` is the constant `42`. Symbol `$x` is no longer needed on this path. On the other path, the value of `b` can be anything but zero. Later, we also discover that the value of `x` is one greater than the original value of `b`. The symbol `$x` is no longer needed on any of the paths, it can be garbage collected.

2.2. **Constraint management in the Static Analyzer.** As mentioned previously, the analyzer collects constraints on symbolic variables encountered in the program to be able to detect if they become unsatisfiable. Solving these constraints is only one side of the coin: generating and managing them is another. Support for constraint management is therefore scattered throughout the analyzer engine. The current solution centers around a solver operating on range-based constraints, which is only capable of handling some common binary operations between symbolic values and concrete integers (called `SymIntExpr`s), and some relational operations between two symbols (`SymSymExpr`s). Although it is very fast, it lacks support for many other commonly used arithmetic operations even on `SymIntExpr`s, such as bitwise operations, multiplication, division, etc.

In 2017, support for an alternative constraint solver backend, the Z3 Theorem Prover, has been added to the engine [9]. Z3 is a state-of-the-art general purpose SMT solver developed by Microsoft Research. Z3 is capable of handling most arithmetic operations unsupported by the current solver, such as those on floating-point values, and it also represents integers more realistically, modeling them with fixed-width bitvectors, granting greater precision in its results.

Unfortunately, the analyzer will not be able to harness the full power of the Z3 Theorem Prover until symbolic expression support is improved in the engine. Namely, the analyzer currently does not build up symbolic expressions consisting of floating-point type values, and subsequently does not generate constraints on them, meaning that information about such expressions never arrives at the constraint manager. Still, without any further effort, Z3 should already be able to improve analysis precision for expressions involving pointers and integers.

Nevertheless, the analyzer still does not employ Z3 as the default constraint solver backend. The reason behind this is its negative impact on the duration of the analysis, with execution times soaring up to and above a factor of 20 times the usual. This slow-down stems from the nature of SMT solvers, which follow complex inner heuristics, and often use up all of the allowed time as limited by the timeout parameter for a single operation. For practical use, an intermediary solution is needed.

One possible compromise is to use the Z3 Theorem Prover for *false positive refutation*. This means to perform the analysis as usual, then post-process the collected bug reports to find those that lie on paths that are found to be infeasible by Z3. This could eliminate a large portion of false positive reports while only introducing a moderate burden on the duration of the analysis.

3. Motivation: an experiment

In an effort to explore how each of the currently available constraint solving backends affect analysis performance and quality, we made the following experiment. For 3 real-world open-source projects, we ran two analyses, each with default settings but differing in the use of the constraint manager backend. We were concerned in the number of reports and execution times in each case. In the table below, the RB keyword denotes the default range-based solver built into the engine, while reports added and removed are meant for the Z3 cases compared to the runs using the range-based solver. Analysis duration is presented in the format `hh:mm:ss`.

Project name	Reports (RB)	Reports (Z3)	Reports removed	Reports added	Duration (RB)	Duration (Z3)
tmux [10]	15	15	0	0	00:01:06	03:09:45
redis [11]	53	20	1	34	00:01:19	03:21:01
xerces-c [12]	69	2	0	67	00:05:40	03:06:22

Table 1. Information about default analyses run using different constraint manager backends on some open-source projects.

It is interesting how the number of bugs seems to drop significantly when using the Z3 backend in the case of *redis* and *xerces-c*. A study of the bug reports could shed light on whether there are already some false positives eliminated, but based on the analyzer statistics shown below for *redis*, it seems more likely that Z3 is timing out and giving up on interesting paths.

Statistic	Range-based	Z3
The # of steps executed.	82 419 176	44 062 305
The # of functions at top level.	19 993	6 834
The # of paths explored by the analyzer.	65 627	33 382
The # of basic blocks in the analyzed functions.	215 656	70 679
The max # of basic blocks in a function.	3 152	1 575
The # of times we reached the max # of steps.	255	163

Table 2. Sample statistics collected by the analyzer about its own operation during the analysis of the *redis* project.

Using the proposed bug post-processing method, the engine will call Z3 significantly fewer times than it would in the case of an ordinary analysis with the Z3 backend. Because of this, it might be reasonable to slightly increase the timeout limit when refutation is switched on, as it could enable the analyzer to explore more paths with Z3, prospectively improving the quality of the analysis.

4. A Prototype of Tiered Constraint Solving

Under tiered constraint solving, we really mean re-solving constraints for paths that lead to bugs. To understand how this can be achieved inside the analyzer technically, we first give a brief overview of the workings of its bug reporting mechanism, and then explain the rationale behind some design decisions of the prototype.

4.1. **Bug reporting in the Static Analyzer.**

If the analyzer finds a critical issue during building the exploded graph, such as a division by zero error, it stops the analysis on that execution path, generates an error node, and emits a bug report (less critical problems would generate a non-fatal error node, in which case the analysis continues on that path, but the bug report is still emitted). Bug reports are continuously collected during analysis, and processed after the construction of the exploded graph is finished.

In order to generate a meaningful path diagnostic from a bug report, and to suppress some reports which are likely to be false positives, the analyzer executes *bug reporter visitors* at this late stage of the analysis. Starting from the error node, visitors travel backwards on the bug path and perform arbitrary operations needed to accomplish their task - usually place additional notes to interesting locations along the path. The bug reporter visitor interface therefore offers a convenient way to implement the tiered constraint solving prototype.

4.2. **Building the refutation visitor.**

Constraints on symbolic values are collected in the program state during the analysis. Whenever a new constraint appears for a symbol, the constraint manager attempts to add it to those already in the state, and if it can prove them to be unsatisfiable, then the current state is said to be infeasible. While building the exploded graph, the engine uses this information in order not to generate a new node for an impossible state. As branching statements are typical points in a program where constraints are added to expressions, constraint management issues relevant for bug report post-processing can be demonstrated on examples involving branching.

Stage 1: Readily available constraints. Sometimes, the range-based constraint solver can reason about a branching condition, and even gets to the point of generating constraints corresponding to a true and a false assumption on the condition, but fails to prove that one of the created states is infeasible. Consider the following example:

```
void g(int d);
void f(int *a, int *b) {
  int c = 5;
  if ((a - b) == 0)
    c = 0;
  if (a != b)
    g(3 / c); // division by zero: false positive
}
```

Arriving at the second `if`, both conditions are understood and translated to ranged constraints, but the solver is not able to prove that they contradict each other. This can be seen from the exploded graph as the current path splits to two, meaning that the constraint manager found both new states to be feasible. On the path assuming that both conditions are true, the exploded node holds the following constraints:

```
Ranges of symbol values:
(reg_$0<int * a>) - (reg_$1<int * b>): { [0, 0] }
(reg_$1<int * b>) - (reg_$0<int * a>): { [-9223372036854775808, -1],
                                         [1, 9223372036854775807] }
```

Here, the `(a != b)` condition has been rearranged to `((b - a) != 0)` by the engine, and the constraint was generated by substracting zero from the full range of possible pointer values, representing the result with a union of the two intervals below and above zero. These constraints can be modeled by the following small `z3` program:

```
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(assert (= (bvsub a b) #x00000000))
(assert (bvslt (bvsub b a) #x00000000))
(assert (bvsgt (bvsub b a) #x00000000))
(check-sat)
(get-model)
```

which, after execution, gives an `unsat` result, i.e. the problem is proved to be unsatisfiable. This is the simplest case our bug reporter visitor should be able to handle: ranged constraints are readily available in the program state, they only need to be fed to a Z3 solver instance in the proper format. For this, constraints on symbolic values need to be converted to the internal expression type used by Z3, which involves the translation of integer relations into their corresponding correct bitvector operations. If the translation succeeds and the Z3 solver can prove the state to be infeasible, the report is marked invalid, and never shown to the user.

Stage 2: Constraints that need to be extracted. If the constraint manager encounters a symbolic expression that it cannot reason about, it also cannot generate a constraint for it. Consider the following example:

```
void g(int c);
void f(int a, int b) {
  int c = 0;
  if (a > 3)
    if (b < 3)
      if (b > a)
        g(5 / c); // division by zero: false positive
}
```

As the constraint manager gives up while interpreting the third `if` condition, it cannot prove that the state where all three conditions are true is not feasible, hence the false positive report. This example is more problematic than the previous one because whenever the constraint manager cannot generate a constraint for an expression, the constraint will also not appear in the program state. The data structure that comes to our aid is the *control flow graph* (CFG), a representation of all paths that might be traversed during program execution. The CFG is constructed by the compiler in an intermediary step of the analysis, and the analyzer core relies on it heavily while building the exploded graph. While the unrecognized condition does not appear among the constraints directly, it can still be extracted from CFG-related information recorded in the exploded node (the terminator statement of the CFG block):

```
Terminator: if b > a
line=6, col=7
Condition: true
...
Ranges of symbol values:
reg_$0<int a>: { [4, 2147483647] }
reg_$0<int b>: { [-2147483648, 2] }
```

from where it could be extracted and fed to a Z3 solver instance with the method outlined at the previous example. Z3 could prove that the path is infeasible, as demonstrated by the program below, which gives an `unsat` result.

```
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(assert (bvsgt a #x00000003))
(assert (bvslt b #x00000003))
(assert (bvsgt b a))
(check-sat)
(get-model)
```

Other examples where false positives can be eliminated with tricks like this can be discovered through systematically designed experiments.

Stage 3: Constraints that need improved symbolic expression support. Symbolic values are the building blocks of symbolic expressions being created by the *symbolic value builder* module during the analysis of a program. Expressions not supported by the symbolic value builder become `UnknownVals` and never get to the point of being handled by the constraint manager. Because of this, such constraints will never appear in the *environment* (the data structure of the program state that maps expressions to their corresponding symbolic values), meaning that they will also not appear in the exploded graph, on which the visitor is meant to operate.

```
void g(int d);
void f(float c) {
  int a = 2;
  if (c > 42.0)
    return;
  if (c > 0.0)
    a = 0;
  if (- 3.14 * c * c > 0)
    g(3 / a); // divide by zero: false positive
}
```

In the above example, `c` is a floating-point value for which the symbolic valuei builder cannot create a valid symbol at the present. As there is no information in the graph that could help prove that the truthness of the third `if` condition leads to an infeasible state, the path leading to the false positive report is created.

This problem can be mitigated by adding support for currently unhandled symbolic values to the symbolic value builder. After such improvements, information needed for the false positive refutation visitor to work will be present in the graph and the previously described methods can be used.

5. Evaluation

In its present state, the refutation visitor implementation is capable of handling constraints that are understood by the default constraint manager and saved into the *range constraints* data structure of the program state. It implements a so-called bug reporter visitor, that is run for each bug report after the construction of the exploded graph is completed. Starting from the error node, the visitor traverses backwards on each buggy execution path, collecting the appropriate ranged constraints from the visited nodes, and adding them

to a Z3 solver instance. At the end of the path, the solver is asked to solve the constraints, and if it finds them unsatisfiable, the bug report gets invalidated.

Evaluation experiments were conducted by running the appropriate analysis configurations on a collection of open-source software projects listed below, on the same virtual machine and using 12 threads. An attempt was made to select both smaller and larger projects written for different purposes in both C and C++ (apart from the previously cited projects: [13], [14], and [15]).

5.1. **Refutation vs. the Z3 constraint solving backend.** The false positive refutation option was designed to provide a compromise between the speed of the default analysis and the precision of an analysis using the Z3 constraint manager backend. We therefore ran analyses with the refutation option switched on on the open-source projects studied in Section 3, in order to compare their results to those using the Z3 backend.

We do not expect the same results for several reasons. First, analyzing projects using the Z3 backend, the whole process uses the Z3 constraint manager, and the resulting exploded graph may differ from the one built in default mode. This means that constraints stored in the graph may be slightly more realistic or precise than those generated in the default mode. However, its working mechanism also differs from the case in which the default analysis is merely enhanced by the refutation visitor. Because of its independent nature, refutation may eliminate false reports that an analysis with the Z3 backend cannot, e.g. those caused by weaknesses in the engine's general operation. And even though it operates on constraints generated by the default solver, the table presented below shows that its advantage in speed may outweigh its disadvantage in granting report quality.

Project name	Reports (default)	Reports (FPR)	Reports (Z3)	Duration (default)	Duration (FPR)	Duration (Z3)
tmux	15	15	15	00:01:06	00:02:02	03:09:45
redis	53	49	20	00:01:19	00:01:22	03:21:01
xerces-c	69	29	2	00:05:40	00:05:50	03:06:22
libWebM	6	6	0	00:00:56	00:00:58	09:26:28
curl	17	14	10	00:01:16	00:01:15	01:34:04
memchached	17	14	1	00:00:37	00:00:38	00:48:32

TABLE 3. Comparison of analyses run with the default configuration, with refutation enabled and using the Z3 constraint manager backend.

5.2. **Refutation vs. default analysis.** From an industry viewpoint, the study of any performance regression refutation poses on the analysis is essential. The following table contains the number of bug reports for two analysis runs for 6 open-source projects, one with a default configuration, and one with the naive prototype of false positive refutation enabled.

As expected, the tiered constraint solving pass did not create any new reports. Since it begins to operate after the exploded graph is completed, it does not participate in the actual analysis process, and can only remove some of the existing reports, but has no means to add new ones. The number of invalidated reports, on the other hand, depends heavily on the analyzed project. In most cases, only a few bugs were removed by the visitor, which is reasonable considering that it currently handles a narrow range of subtle false positive cases. The *xerces-c* project stands out in this regard, with more than a half of its bugs thrown away. After performing a manual inspection of some of the removed reports, either the falseness of the reports was difficult to determine (because of long bug paths), or we found that the report was truly a mistake on the analyzer's behalf and its removal increased the overall quality of the analysis.

Project name	Reports (default)	Reports (FPR)	Reports removed	Duration (default)	Duration (FPR)
tmux	15	16	0	00:01:01	00:01:18
redis	53	161	4	00:02:15	00:04:01
xerces-c	69	40	40	00:03:22	01:01:22
libWebM	6	28	0	00:01:21	00:02:50
curl	17	36	3	00:01:01	00:01:00
memcached	17	14	3	00:29:30	00:40:17

TABLE 4. Report numbers for analyses with a default configuration and with false positive refutation (FPR) enabled for some open-source projects.

6. FUTURE WORK

Although the tiered constraint solving protoytype is already capable of eliminating some false positive bug reports, its functionality can be extended and its results greatly improved once further enhancements outlined below will be introduced into the analyzer.

6.1. **Symbolic expression support.** The main purpose of the initial introduction of the Z3 backend was to enable the analyzer to reason about floating-point values. For this to work fully, the analyzer needs to generate and handle symbolic floating-point expressions (`SymFloatExpr`s and `FloatSymExpr`s). Apart from floats, symbolic expression support should generally be improved in the symbolic value builder module, including arithmetic operations involving values other than concrete integers. This could fundamentally improve the precision of the analysis.

The analyzer sometimes makes assumptions about algebraic operations that were written with the integer-based constraint manager in mind, and often do not hold for other types of values (e.g. the `x == x` is `true` assumption does not hold for special floating-point values like `NaN`s). Along with the introduction of new types, these assumptions could be revised and extended to support operations defined for any new types, for example for floats. Additionally, assumptions like these could also be checked for expressions involving an integer and a symbol or two symbols.

Code parts responsible for dropping constraints that will not be digested by the symbolic value builder are scattered around in its current form. This makes it difficult to evaluate how changing the level of detail affects the performance of the engine (and it is also more difficult to determine what is handled). This logic could be collected to one place behind a flag, so that symbolic expression handling could be controlled easily.

6.2. **Packaging.** Although the adoption of the tiered constraint solving solution in the Clang Static Analyzer is already in progress, its usage for a normal user is hindered by packaging issues. Users typically use the analyzer as part of their continuous integration toolchains and are reluctant to make modifications to their command scripts, so Z3 support should be granted just by updating their *clang* to the latest version.

This is however not possible because of the project's licensing policy. Although Microsoft Research open-sourced their Z3 theorem prover, its license is still not compatible with *clang*'s liberal open-source license, and thus cannot be included in the latest *clang* package. On the contrary, the Z3 sources need to be downloaded and installed separately by each user and then *clang* needs to be built with special flags that find the Z3 installation and enable its support. Most of the users would probably abandon the refutation feature because of such inconveniences.

One idea to solve this situation would be to find and integrate another SMT solver like the Z3 theorem prover, but with a compatible license. Alternatively, if no project is found that suits the analyzer's needs, a small SMT solver could

be re-implemented inside the analyzer much like the range-based solver, but a more powerful one.

To facilitate solver comparing experiments and to make solver backend switching more flexible, a general SMT solver interface could be implemented in the analyzer. The current constraint management framework relies heavily on the built-in range-based solver, and only has been extended to support Z3 in a very special manner. Most of the duplicate work involved in adding a new backend at the present could be avoided with a general interface.

7. Conclusion

Minimizing the number of false positive reports is a critical issue for most static analysis tools in order to grant high-quality results to their users. A method taking an important step towards this goal was presented for one of the most widely used open-source static analysis tools for C family languages, the Clang Static Analyzer. The solution works by introducing an additional step towards the end of the analysis, when constraints on symbolic expressions encountered on the buggy execution path are re-evaluated by a powerful external constraint solver engine, invalidating a bogus bug report if the path leading to it is found to be infeasible. This step is needed because the default built-in constraint solver is designed to prefer speed over precision, while using the precise external solver for the whole process would result in unacceptably long execution times. In order for the analyzer to retain its industrial-strength performance, a practical intermediary solution was needed.

The tiered constraint solving solution described in this paper is careful to preserve close-to-usual execution times while eliminating many of the false positive reports, as found by an evaluation on a set of real-world software projects. Additionally, an agenda of possible enhancements was outlined that might be useful to study and implement to further improve the results. The prototype is currently under review by the open-source community.

8. Acknowledgement

We owe our special thanks to Artem Dergachev and George Karpenkov, core developers of the Clang Static Analyzer, for the discussion and advice regarding the proposed changes.

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

References

[1] Clang Static Analyzer. https://clang-analyzer.llvm.org
[2] HAMPAPURAM, Hari; YANG, Yue; DAS, Manuvir. Symbolic path simulation in path-sensitive dataflow analysis. In: ACM SIGSOFT Software Engineering Notes. ACM, 2005. p. 52-58.
[3] DE MOURA, Leonardo; BJØRNER, Nikolaj. Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2008. p. 337-340.
[4] "clang" C language family frontend for LLVM. https://clang.llvm.org/
[5] RICE, Henry Gordon. Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society, 1953, 74.2: 358-366.
[6] REPS, Thomas; HORWITZ, Susan; SAGIV, Mooly. Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1995. p. 49-61.
[7] XU, Zhongxing; KREMENEK, Ted; ZHANG, Jian. A memory model for static analysis of C programs. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Springer, Berlin, Heidelberg, 2010. p. 535-548.
[8] Artem Dergachev: Clang Static Analyzer - A Checker Developer's Guide. 2016. https://github.com/haoNoQ/clang-analyzer-guide
[9] Dominic Chen: Add new Z3 constraint manager backend. Differential Review. 2017. https://reviews.llvm.org/D28952
[10] Tmux, a terminal multiplexer. https://github.com/tmux/tmux/
[11] Redis, an open source, in-memory data structure store. https://redis.io/
[12] Xerces-C++ XML Parser. https://xerces.apache.org/xerces-c/
[13] WebM, an open web media project. https://www.webmproject.org/
[14] Curl, a command line tool for transferring data with URLs. https://curl.haxx.se/
[15] Memcached, a distributed memory object caching system. https://memcached.org/

Email address: rekanikolett@caesar.elte.hu

Eötvös Loránd University, Department of Programming Languages and Compilers, Pázmány Péter st. 1/C., Budapest, Hungary

Email address: xazax@caesar.elte.hu

Eötvös Loránd University, Department of Programming Languages and Compilers, Pázmány Péter st. 1/C., Budapest, Hungary