

# A Case Study on the Quality of Static Analysis Bug Reports

Kristóf Umann<sup>1</sup>, Zoltán Porkoláb<sup>1</sup>

<sup>1</sup>ELTE, Eötvös Lóránd University, Budapest, Hungary. Faculty of Informatics, Department of Programming Languages and Compilers

## Abstract

Finding bugs in software using automated tools has enjoyed generous attention for as long as humans wrote software. With the increase in computing capacity and advancement in compiler theory, static analyzer tools like the Clang Static Analyzer have been an unexpandable part of several software development projects. The Clang Static Analyzer, like many other tools of its kind, employs heuristics, which may lead to reports on correct code – so-called *false positives*. The evaluation and the potential fixing of the reports cannot be automated, requiring valuable time from the most expensive and least available resource during development, a human expert. While the *finding* of bugs stands in the focus of academic research, the intelligible presentation of those findings to the user was less frequently discussed. In our paper, we survey several reports emitted by the Clang Static Analyzer to understand what makes a bug report hard to understand. Our study aims to pave the way for further research to fix the problems discussed here.

## Keywords

static analysis, C, C++, Clang, LLVM, bug report quality

## 1. Introduction

Maintenance costs take a considerable chunk out of the budget of any development team. Most of these expenses are spent fixing bugs. The earlier a bug is detected, the lower the cost of the fix [1]; therefore, alongside traditional methods of testing, it is worth pursuing automated tools, such as static analyzers.

Static analyzers provide early feedback on the quality of software by design. It is a popular method for finding bugs and code smells, and various tools implement it [2]. Many of the applied techniques are fast and cheap enough to be integrated into the Continuous Integration (CI) loops, therefore, they have a positive impact on the speeding up of development.

Most static analysis methods apply heuristics, meaning they may often *underestimate* or *overestimate* program behaviour [3]; in other words, static analysis *trades precision for coverage*. This means that after the analysis, all reports must be inspected by a professional who has to decide manually whether the report stands or is a false positive. This further strains what is usually the

---

SQAMIA 2023: Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 10–13, 2023, Bratislava, Slovakia


✉ szelethus@inf.elte.hu (K. Umann); gsd@inf.elte.hu (Z. Porkoláb)

🌐 gsd.web.elte.hu (Z. Porkoláb)

🆔 0000-0002-6679-5614 (K. Umann); 0000-0001-6819-0224 (Z. Porkoláb)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

most expensive and the least available resources. It has the utmost importance to maximize the effectiveness of the step where humans are involved [4]. As such, the communication between the automated analysis tool and the user must be as smooth as possible.

In this paper, we survey bug reports generated by the Clang Static Analyzer, inspect various selected examples, and discuss where communication with the user is lacking. Alongside an evaluation of how many reports are intelligible from the complete report set, we highlight two specific faults observed in several reports.

## 2. Measurement methodology

In this section, we discuss the way we approached surveying the Clang Static Analyzer’s reports.

### 2.1. Briefly on how the analyzer works

A decent simplified explanation of how symbolic execution [5, 6] works, which is the technique used by the Clang Static Analyzer, would be the following. Suppose we have a C++ interpreter. The interpreter’s input is the source code (written in one of the C family of languages). Starting from the primary function, it reads and executes each statement one by one, sometimes by jumping to another function. It keeps track of the values of variables (using *symbolic* [7], as opposed to concrete values) and the call stack. If it encounters a condition, it splits the analysis in two – one where the condition is assumed true and one where it is assumed false [8]. Before executing a statement, it allows various modules, or so-called “checkers”, to check preconditions, and after the execution, it applies postconditions. If any of these conditions are violated, it emits a warning for the user.

The Clang Static Analyzer [9, 10, 11, 12] is an open-source tool that implements symbolic execution on the C family of languages. Clang [13], which encompasses the static analyzer, is the primary compiler frontend for LLVM and has now enjoyed more than 15 years as one of the leading static analyzers for C/C++. For the remainder of the paper, we refer to the Clang Static Analyzer under the term *analyzer*.

### 2.2. Our focus on the measurements

Following the above analogy, the interpreter is the analyzer’s engine, and checkers define pre- and postconditions. For instance, `core.DivByZero` defines that the denominator must not be 0. If the engine can prove that the denominator is 0, a warning is emitted; otherwise, it is assumed that it is non-zero. Some checkers implement more sophisticated semantics – `unix.MismatchedDeallocator` keeps track of whether `malloc()` or `operator new` was allocated some memory and whether the corresponding deallocator was used to release it.

The former example is a checker that relies strongly on the correct operation of the engine. While the latter example relies on the engine as well, its own bookkeeping is a large component of its logic. For this reason, we chose to limit our investigations to `core.DivByZero`, because we are more interested in the general mechanics and faults of bug report generation and less so in reports on specific domains.

	Total reports	Acceptable	Not enough info	Incomprehensible
Acid	1	1	0	0
ffmpeg	6	5	1	0
LLVM + Clang	11	8	2	1
OpenSSL	1	1	0	0
postgres	2	1	0	1
QTBase	7	2	1	4
Vim	2	2	0	0
Xerces	1	0	1	0
Total	31	20	5	6

**Table 1**

A partial summary of our findings. Each row contains how many bug reports surveyed for the given project, and their intelligibility.

### 3. Results & Discussion

We tested the latest version of the analyzer<sup>1</sup> on the following open-source C and C++ projects: Acid [14], ffmpeg [15], LLVM and Clang [16], OpenSSL [17], postgres [18], QTBase [19], vim [20] and Xerces [21]. Combined, these projects cover a wide variety of coding techniques, codebase sizes, and different versions of the languages' standards.

Table 1. partially summarizes our findings. We sorted the results we surveyed into the 3 categories:

- *Acceptable*: It is possible to understand the report, and whether it stands, even if it could be improved.
- *Not enough info*: It is *not* possible understand the report, but it is possible to say which function calls / value changes the analyzer neglected to explain, and a domain expert may possess the missing information and judge whether the report stands.
- *Incomprehensible*: The entire bug report is incomprehensible, and its doubtful that even a domain expert can judge the report.

Overall, we found that out of 31 reports, 11 division by zero reports were not acceptable. Of that, 6 reports were incomprehensible; 4 was reported in QTBase, and concerned code where numerous integers and floating point numbers were used in a hard-to-follow manner. An example is shown on Figure 1. For the other reports, it was more apparent that the analyzer was at fault for failing to generate a proper report, rather than the source code being messy. We feel that the analyzer was also the tool to blame for the reports that lacked more information. An example for this case can be seen in Figure 2.

Out of 31 reports, only 20 reports were at all acceptable. That is not to say that extracting the necessary information was easy but at the very least possible. We identified a pattern of *indirect reasoning*, shown in Figure 3. The example shows that `i`'s value is 0 when we assume it to be equal to `NumElements`, which is why the division-by-zero warning occurs at the remainder operator.

<sup>1</sup>The latest version of Clang at of time of writing is 16.0.6.

While the example is only a few lines, we found that this indirect reasoning in other reports is often spread over several function calls, several hundred lines of code, and several logical statements that wear on the memory of the user. It would be easier to understand this report if the warning stated “NumElements is known to be 0 because it is equal to i, which has a value of 0.”.

Supporting findings of a previous research [22], we also identified that many of the leading notes in the reports are superfluous. Of the acceptable reports, 6 reports had fewer than 10 notes, 8 had 10 to 19 notes, 6 had 20 or more notes. We measured how many notes we could omit in sequence from the first note – we found that only the last few are usually important. Of the acceptable reports, 11 reports had less than 10 meaningful notes, and the rest had fewer than 20.

```

Center at the top.
if (y2 < y1) {
  37 < Assuming 'y2' is < 'y1' >
  // yC <= y2 < y1
  // Long right edge.
  if (yC != y2) {
    38 < Assuming 'yC' is equal to 'y2' >
    d2 = centerFrac * value / (v2->y() - center->y());
    dd2 = ((value <= 8) / (v2->y() - center->y()));
    fillLines<clip, TopDown, LeftToRight>(bits, width, height, yC, y2, x2, dx2,
                                          x1, dx1, d2, dd2, dd);
  }
  dx2 = ((v1->x() - v2->x()) <= 8) / (v1->y() - v2->y());
  39 < Division by zero
  < For more information see the checker documentation.
  x2 = v2->x() + v2Frac * (v1->x() - v2->x()) / (v1->y() - v2->y());
  fillLines<clip, TopDown, LeftToRight>(bits, width, height, y2, y1, x2, dx2,
                                          x1, dx1, value, 0, dd);
}

```

**Figure 1:** Example of a report that is incomprehensible from `qdistancefield.cpp` in `QTBase`. The number of small variables make it hard to understand how the values of variables change.

```

static AVFrame *get_palette_frame(AVFilterContext *ctx)
{
  AVFrame *out;
  PaletteGenContext *s = ctx->priv;
  AVFilterLink *outlink = ctx->outputs[0];
  double ratio;
  int box_id = 0;
  struct range_box *box;
  /* reference only the used colors from histogram */
  s->refs = load_color_refs(s->histogram, s->nb_refs);
  if (!s->refs) {
    1 < Assuming field 'refs' is non-null >
    av_log(ctx, AV_LOG_ERROR, "Unable to allocate references");
    return NULL;
  }
  /* create the palette frame */
  out = ff_get_video_buffer(outlink, outlink->w, outlink->h);
  if (!out) {
    2 < Assuming 'out' is non-null >
    return NULL;
  }
  out->pts = 0;
  /* set first box for 0..nb_refs */
  box = &s->boxes[box_id];
  box->len = s->nb_refs;
  box->sorted_by = -1;
  box->color = get_avg_color(s->refs, box);
  3 < Calling 'get_avg_color' >
}

static uint32_t get_avg_color(struct color_ref * const *refs,
                             const struct range_box *box)
{
  4 < Entered call from 'get_palette_frame' >
  int i;
  const int n = box->len;
  uint64_t r = 0, g = 0, b = 0, div = 0;
  5 < 'div' initialized to 0 >
  for (i = 0; i < n; i++) {
    6 < Assuming 'i' is <= 'n' >
    7 < Loop body executed 0 times >
    const struct color_ref *ref = refs[box->start + i];
    r += (ref->color >> 16 & 0xff) * ref->count;
    g += (ref->color >> 8 & 0xff) * ref->count;
    b += (ref->color & 0xff) * ref->count;
    div += ref->count;
  }
  r = r / div;
  8 < Division by zero
  < For more information see the checker documentation.
}

```

**Figure 2:** Example of a report that has missing information in `vf_palettegen.c` from `ffmpeg`. `div`'s value depends on `n`, which in turn got its value from `box->len`, that got its value from `s->nb_refs`. However, the analyzer never explained why it thinks it is entitled to assume that this value can be 0.

```

int Indices[8];
for (unsigned i = 0; i != NumElts; ++i)
    Indices[i] = i;
for (unsigned i = NumElts; i != 8; ++i)
    Indices[i] = NumElts + i % NumElts;

```

9 < Assuming 'i' is equal to 'NumElts' >  
10 < Loop body executed 0 times >  
11 < Entering loop body >  
12 Division by zero  
For more information see the checker documentation.

**Figure 3:** Example of a report where it is only indirectly proved that the denominator is 0 in `AutoUpgrade.cpp` from LLVM. `i`'s value is 0 when we assume it to be equal to `NumElts`, which is why the division of zero occurs at the remainder operator.

## 4. Conclusion

Symbolic execution, a static analysis technique, is a powerful tool to find deeply rooted programming errors. Despite its strengths, it often emits bug reports that leave much to be desired, demanding even more of the most expensive resource and least available in a software development project: human experts.

We surveyed a popular static analyzer tool, the Clang Static Analyzer, that checks code of the C family of languages with a complex technique called symbolic execution. We discussed that it consists of an engine that interprets the code and several checkers defining programming semantics. We took measurements of a checker that does relatively little modelling and relies mainly on the engine and generalized bug reporting techniques.

We found that out of 31 reports, 6 are incomprehensible, 5 do not contain enough information to understand the finding, and 20 are acceptable, even if poor. We identified two specific classes of problems: one where chains of logical statements need to be deciphered by the user to understand the report, and one where only the last few notes of the report were actually important.

## References

- [1] B. Boehm, V. R. Basili, Software defect reduction top 10 list, *Computer* 34 (2001) 135–137. URL: <http://dx.doi.org/10.1109/2.962984>. doi:10.1109/2.962984.
- [2] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler, A few billion lines of code later: Using static analysis to find bugs in the real world, *Commun. ACM* 53 (2010) 66–75. URL: <http://doi.acm.org/10.1145/1646353.1646374>. doi:10.1145/1646353.1646374.

- [3] M. Anders, I. S. Michael, Static program analysis., 2012. URL: <https://users-cs.au.dk/amoeller/spa/spa.pdf>.
- [4] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in: 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 672–681. doi:10.1109/ICSE.2013.6606613.
- [5] C. Szabó, M. Kotul, R. Petruš, A closer look at software refactoring using symbolic execution, in: Proceedings of the 9th International Conference on Applied Informatics, Volume 2, Eszterházy Károly College, 2015. URL: <https://doi.org/10.14794/icai.9.2014.2.309>. doi:10.14794/icai.9.2014.2.309.
- [6] E. Fülöp, N. Pataki, A DSL for resource checking using finite state automaton-driven symbolic execution, Open Computer Science 11 (2020) 107–115. URL: <https://doi.org/10.1515/2Fcomp-2020-0120>. doi:10.1515/comp-2020-0120.
- [7] Z. Xu, T. Kremenek, J. Zhang, A memory model for static analysis of C programs (2010) 535–548. URL: <http://dl.acm.org/citation.cfm?id=1939281.1939332>.
- [8] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, ACM Comput. Surv. 51 (2018).
- [9] Clang SA, Clang Static Analyzer, 2019. <https://clang-analyzer.llvm.org/>.
- [10] Checker Developer Manual, Clang Static Analyzer: Checker Developer Manual, 2019. [https://clang-analyzer.llvm.org/checker\\_dev\\_manual.html](https://clang-analyzer.llvm.org/checker_dev_manual.html) (last accessed: 28-02-2019).
- [11] A. Dergachev, Clang Static Analyzer: A Checker Developer's Guide, 2016. <https://github.com/haoNoQ/clang-analyzer-guide> (last accessed: 28-02-2019).
- [12] A. Zaks, J. Rose, Building a checker in 24 hours, 2012. <https://www.youtube.com/watch?v=kdxlsP5QVPw>.
- [13] C. Lattner, Llm and clang: Next generation compiler technology, 2008. Lecture at BSD Conference 2008.
- [14] Matthew Albrecht, Acid game engine, 2023. URL: <https://github.com/EQMG/Acid>.
- [15] F. Developers, Ffmpeg, 2023. <https://github.com/FFmpeg/FFmpeg>.
- [16] C. Lattner, V. Adve, Llm: A compilation framework for lifelong program analysis & transformation (2004) 75.
- [17] OpenSSL Software Foundation, Openssl, 2022. URL: <https://openssl.org/>.
- [18] postgres Developers, postgres, 1986. <https://github.com/postgres/postgres>.
- [19] QTBase developers, Qtbase, 2023. URL: <https://github.com/qt/qtbase>.
- [20] A. Robbins, E. Hannah, L. Lamb, Learning the vi and vim editors, " O'Reilly Media, Inc.", 2008.
- [21] Apache Software Foundation, Apache xerces, 2022. URL: <https://xerces.apache.org/>.
- [22] T. Brunner, P. Szécsi, Z. Porkoláb, Bug path reduction strategies for symbolic execution, in: THE 11TH CONFERENCE OF PHD STUDENTS IN COMPUTER SCIENCE, 2018, p. 159.