13th International Conference on Applied Informatics
Eszterházy Károly Catholic University
Eger, Hungary, February 19–21, 2026

Static Analysis checkers almost a decade later: Where are they now?

Kristóf Umann, Zoltán Porkoláb

ELTE Eötvös Lóránd University, Budapest, Hungary. Faculty of Informatics,
Department of Programming Languages and Compilers
szelethus@inf.elte.hu, gsd@inf.elte.hu

1. Introduction

Uninitialized variables have been a source of errors since the beginning of software engineering. Some programming languages (e.g. Java and Python) will either force initialization or zero-initialize automatically, but others, like C and C++, will leave the state of uninitialized non-static variables undefined. Dynamic analyzers specifically tailored to find use-before-initialization errors have existed for decades, but they require the problematic code to be executed. In contrast, static analyzers do not, and they also enjoy some other unique adventages. With the advance of techniques like symbolic execution, they can also detect deep rooted initialization errors.

In 2018, we created a checker in the Clang Static Analyzer to find improperly initialized object, including an array of heuristics to keep the volume of false positives in check. We published our findings in a paper [3] with measurements on large open-source projects. In the years since, thousands of commits landed affecting the symbolic execution engine in Clang, and literally hundreds of thousands into LLVM, some of which might have indirect impact on analysis results.

This paper investigates how the results from a static analysis checker changes as the codebase around in matures in over 12 releases spanning 8 years.

2. Measurement configurations

As false positives are consistently identified as among the greatest pain points to adaption [1, 4, 5], our implementation also takes serious mitigating measures

against them.

While correctly identified and reported uninitialized objects are *by definition* true positives, the lack of initialization can be intentional to gain performance when meaningful default values cannot be given. Reporting intentenionally uninitialized objects is invaluable, and are *perceived* by users as false positives.

For this reason, we made our checker configurable, allowing the user to enable, disable or fine-tune some of our heuristics for a particular project.

Pedantic: Through testing our prototype, we observed that objects that do not initialize a single one of their fields are often created intentionally. However, this heuristic can result in a higher amount of false negatives than maybe desired, so we made it toggleable through an option called *Pedantic*, and is disabled by default.

Pointer chasing: Indirection raises a philosophical question: Is an object responsible for leaving its pointee object in a fully deterministic state? One perspective we could take is to guess whether the objects *owns* the pointee. However, ownership is a conceptually popular, but non-standardized concept within C++ [2]. We created an option called *pointer chasing* to check for the initialization of pointees but it is disabled by default.

Guarded field analysis: Much like for tagged unions, where an enumeration value is checked to guard against accessing an inactive union data member, it is a common idiom to guard against uninitialized variable misuse. We implemented a primitive heuristic to identify guarded fields using abstract syntax tree analysis, which is computationally less demanding but not as robust as symbolic execution. As with the others, this option is off-by-default.

3. Preliminary results

We are assembled a suite of 8 open-source C++ projects of varying size and complexity: LLVM+Clang, QTbase, protobuf, libwebm, xerces, openrct2, bitcoin and tinyxml2. On version 21.0.0, we ran the Clang Static Analyzer on these projects with different configurations to our checker. We measured how enabling one parameter, while leaving the others disabled, affected the analysis result. We summarize our early results in Table 1, considering only issues found by UninitializedObjectChecker.

In the full paper, we will compare the result sets of the Clang Static Analyzer on the same suite, but on version 9.0.0, the first to feature our checker. As even seemingly small changes can change the behavior of symbolic execution, we expect our

4. Conclusion

Uninitialized objects are a common source of program instabilities, and have been in the interest of automated tools for decades. Our paper investigates how a static

Table 1. Findings from UninitializedObjectChecker from the Clang Static Analyzer version 21.0.0.

Project	Default	Pedantic	Pointee	Guarded
LLVM+Clang	249	251	385	240
qtbase	51	56	71	51
protobuf	26	27	40	26
libwebm	5	5	5	5
xerces	1	1	1	1
Others	0	0	0	0

analysis solution, UninitializedObjectChecker from the Clang Static Analyzer, finds such errors when it was first released, and 12 releases later.

Acknowledgment

Project no. C2314106 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the KDP-2023 funding scheme.

References

[1] M. Christakis, C. Bird: *What developers want and need from program analysis: An empirical study*, in: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 332–343.

[2] G. Horváth, N. Pataki: *Categorization of C++ Classes for Static Lifetime Analysis*, in: Proceedings of the 9th Balkan Conference on Informatics, ACM, Sept. 2019, 21:1–21:7, doi: 10.1145/3351556.3351559, url: https://doi.org/10.1145%2F3351556.3351559.

[3] K. Umann, Z. Porkoláb: *Detecting Uninitialized Variables in C++ with the Clang Static Analyzer*, Acta Cybernetica 25.4 (Nov. 2020), pp. 923–940, issn: 0324-721X, doi: 10.14232/actacyb.282900, url: http://dx.doi.org/10.14232/actacyb.282900.

[4] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, H. C. Gall: *Context is king: The developer perspective on the usage of static analysis tools*, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 38–49, doi: 10.1109/SANER.2018.8330195.

[5] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, A. Zaidman: *How developers engage with static analysis tools in different contexts*, Empirical Software Engineering 25.2 (Nov. 2019), pp. 1419–1457, issn: 1573-7616, doi: 10.1007/s10664-019-09750-5, url: http://dx.doi.org/10.1007/s10664-019-09750-5.