Optimizing Static Analysis Runtime via Profile-Guided Optimization

Kristóf Umann[1], Zoltán Porkoláb[1]

[1]ELTE, Eötvös Lóránd University, Budapest, Hungary. Faculty of Informatics, Department of Programming Languages and Compilers

Abstract

Static analysis is a well established technique to find programming errors. Unlike dynamic analyzers, most static analyzers do not require precise input values to run and can achieve much greater code coverage. With that said, dynamic tools have much better precision in a given path of execution, and can find deep-rooted programming errors, something that is much more difficult in the absence of input values. Path-sensitive analyses, and in particular, symbolic execution mimics dynamic behaviour at compile-time, making it capable to also find such errors.

A great advantage of static analyzers is that they provide an early feedback on the quality of the software, as they can be integrated in the early stages of the CI pipelines. However, tools like the Clang Static Analyzer, which employs symbolic execution can be many times more resource intensive than compilation. In extreme cases, analyzing large codebases can take as long as 8 hours.

In this paper, we discuss how Profile-Guided Optimization (PGO) can be used to complement traditional, static optimization techniques to shorten the analysis time of the Clang Static Analyzer. As this requires gathering profiling data during the analysis of projects, we also discuss profile selection strategies. Our results show that regardless of profile, analysis times shrank by at least 12%, while optimal profile can show up to 17% improvement.

Keywords

static analysis, profile-guided optimization, LLVM, Clang Static Analyzer

1. Introduction

Today, programming errors and vulnerabilities in software are a greater issue than ever before. Such issues can cause serious material damage or even can threaten human life. To avoid such situations software systems are intensively analyzed by various tools. Among these methods, static analysis is one of the most promising, rapidly developing area with a number of open source and proprietary tools [1, 2, 3, 4].

Most forms of static analysis works on the source code to detect various properties of the system, including possible errors, without running the program. Therefore, static analysis does not require the complete running environment (network, databases, etc.), and does not depend on carefully selected data to cover the most interesting parts of the software. As a consequence, static analysis is usually applicable much earlier than the other software checking methods, many cases available even during the early stages of the development process. As the earlier the bug is detected, the cheaper it is to fix it [5], static analysis is a perfect fit method for the development-bug detection-bug fixing cycle. Symbolic execution [6, 7, 8] in particular is a hot topic in static analysis.

Loosely speaking, the idea behind symbolic execution is similar to a (dynamic) language interpreter. The interpreter's input is usually the source code. 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 and manages a call stack. Unlike language interpreters, if it encounters a condition, it forks the analysis – one fork where the condition is assumed true and one where it is assumed false. This leads to a combinatorial explosion according to the number of program states, which provides more

SQAMIA 2026: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 16–18, 2026, Niš, Serbia

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)



© 2026 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 ISSN 1613-0073

published 2026-11-19

CEUR-WS.org/Vol-4336/short3.pdf

information about runtime behavior, but also makes the analysis rather expensive [9]. While there are algorithmic approaches to keep the cost of path explosion tolerable [10], they usually result in loss of precision [11]. Regardless of such solutions, the resulting analysis binary should be optimized to speed up its execution, and preferably with techniques beyond the traditional compiler optimizations [12].

Profile-guided optimization (PGO) emerged in the 1980s [13] to go beyond static optimization techniques. Profile-guided optimization of a program is done in the following major steps: [14]

1. Create an instrumented binary capable of emitting what parts of the program are executed at runtime and how many times;
2. Run the instrumented program to gather runtime information. The collected data is called the *profile data*;
3. Compile the program again while supplying the profile data to the optimizing compiler.

The resulting profile-optimized binary is frequently reported to run 10-30% faster than without PGO [14].

Profile data is inherently specific to how the data was collected. For instance, Clang, a popular C++ compiler, has a lengthy guide to collect profile data on compiling Clang itself – the resulting profile-optimized Clang binary will be specifically tailored to compile the LLVM and Clang codebases. Project-specific PGO compilers as frequently discussed, but not how other related compiler tools such as the Clang Static Analyzer behave with regards to PGO.

In this paper, we investigate how much efficiency gain can be realized on the notoriously slow Clang Static Analyzer using profile-guided optimization. We also measure how different profiles affect analysis performance, e.g. whether there is difference in between analysis times on a C++ project if the analyzer tool was profiles on a C or C++ analysis.

This paper is structured as follows. In Sections 2 and 3 we detail the technical background for symbolic execution and profile-guided optimization respectively. In Section 4 we show our measurement method, and in Section 5 we show and evaluate our results. The paper concludes in Section 6.

2. Symbolic execution

Concrete execution, what we would consider the "normal" execution of the program, is done by a specific input set to exercise a single path of execution. In contrast, most forms of symbolic execution need no input values and explore multiple paths of execution, covering entire classes of inputs [15]. These input values, and runtime variables of the program are assigned *symbolic* (rather than concrete) values. An analysis engine models the program behavior with a *store*, which is a mapping of variables to symbolic values, and *path constraints*, which contains constraints on symbolic values [15, 16]. The store is updated when a memory location is written, and the constraint solver is updated after each evaluation of a conditioned branch.

2.1. Clang Static Analyzer

The Clang Static Analyzer [17, 18, 19, 20], the tool of choice for most of our research, is an open-source tool that implements symbolic execution on the C family of languages. Over the course of a decade, it grew to be regarded as a stable and reliable tool for academic and industrial [21, 22] purposes, and is still actively maintained. It enjoys a variety of advantages of being built directly into the Clang compiler, as the abstract syntax tree (*AST*) and control-flow graph it implements are battle tested, considering that they are also used for compiling millions of lines of production code. Clang [23] is a compiler front-end for LLVM, the umbrella project that offers a wide selection of algorithms that are useful for optimization.

The Clang Static Analyzer uses symbolic execution to find programming errors using so-called "checkers" [24, 25, 26, 27]. Before modeling (or "interpreting") a statement, each checker may check preconditions (e.g. is the denominator known to be 0 in this division?), and after the execution, it applies

postconditions (e.g. if it can not be proven that the denominator is 0, assume from this point that it is non-zero). If any of these conditions are violated, checkers emit a warning.

3. Profile-Guided Optimization

Profile Guided Optimization (PGO) adds the missing link to “"traditional"” static analysis, which is an insight on the program’s and dynamic behavior, enabling compilers to optimize accordingly. Based on execution counts, branching statistics, and memory access profiles the optimizing compiler reorder, inline, and lay out code accordingly. PGO can yield speedups often in the range of 5%-30% [28, 29, 13].

LLVM implements an infrastructure to optimize Clang via instrumentation. This is done by building a profiling compiler that collects run-time data. The two principle technique behind this is edge profiling [30] and path profiling [31].

Edge profiling records the number of times a specific edge is taken in the control-flow graph, yielding branching behaviors. This can be used to identify so-called *hot and cold branches*, allowing the optimizing compiler to later to reorder basic blocks to improve instruction cache locality or improve the branch-prediction heuristics.

Path profiling counts the number of times each path from entry block to exit block is executed in the control-flow graph, capturing contextual information such as how different loop-exit paths are taken. Although path profiling is more complex and expensive than edge profiling, it provides a more complete characterization of program behaviors. The combination of these two techniques enable optimization such as code-layout adjustment and change inlining decisions based on actual execution patters [13].

4. Measurement method

In this section, we discuss our measurement method.

4.1. Profile generating and compilation

Clang’s documentation for PGO builds [32] intends to create a compiler binary that is optimized for the compilation of LLVM and Clang itself. This is done in five steps:

1. Build Clang in release configuration, which includes all the “standard” optimizations. This is called the stage 1 build.
2. Using the just built stage 1 Clang, build Clang again, configured to emit profiling data in `.profraw` formatted files. This is the stage 2 build.
3. Using Clang in stage 2, compile Clang yet again. During this process, several `.profraw` files are generated.
4. Use `llvm-profdata` from the stage 1 build to merge all `.profraw` files into a single `.prof` file.
5. Using Clang in stage 1 and the merged profile data in the previous step, compile Clang one last time. This will yield a Clang binary built with profile-guided optimization, sometimes called the stage 3 build.

Naturally, one can build a different codebase to generate profile data, which would yield a stage 3 compiler that is specifically optimized to build that codebase instead. However, our aim is to decrease analysis speed, not compilation.

For this reason, in step 3, ran the Clang Static Analyzer instead, which must be invoked from the Clang binary with `-analyze`. In order to scale up our measurements to real-world projects, we used CodeChecker [22] to generate compilation databases and run the analysis with the stage 2 Clang profiler. We followed the rest of the steps – merging the raw profile data and building the stage 3 compiler – as the official documentation recommends.

We built two stage 3 compilers on LLVM release 22.1.8, each with one profile data. We generated the profiles by analyzing the following projects:

1. FFmpeg, which is command line toolbox to manipulate, convert and stream multimedia content written in C. It is a medium sized project that takes roughly 5 minutes to analyze with the Clang Static Analyzer. We call the resulting stage 3 compiler *Clang-FFmpeg*.

2. LLVM and Clang, which is a massive C++17 codebase that takes around 85 minutes to analyze with the Clang Static Analyzer. We call the resulting stage 3 compiler *Clang-LLVM*.

4.2. Data gathering

We chose two C and two C++ projects to run our measurement data on: GNU binutils v2.46.0 [33], FFmpeg n7.1.1 [34], LLVM and Clang 20.1.1 [35], Xerces v3.3.0 [36]. LLVM is a massive codebase which we included to show PGO behaves on projects of such scale, while binutils, FFmpeg, LLVM and Xerces are medium sized.

In order to conduct paired statistical tests later, we created a benchmark that analyzed each project five-five times interchanging between either stage 1 Clang and Clang-FFmpeg. We repeated the paired analyses measurement by analyzing with stage 1 Clang and Clang-LLVM five-five times as well. As static analysis is usually done in parallel in a production environment, we chose to use all available threads on our machine.

Our workbench had an AMD EPYC 9354 processor, logically restricted to 8 cores at 3.2 GHz with 16 available threads, and 64 GB of memory.

Project	Profile	Non-PGO (s)			PGO (s)			Speedup
		Mean	Med	SD	Mean	Med	SD	
FFmpeg	`Clang-FFmpeg`	300.71	303.47	6.49	251.87	249.30	5.49	**16.24%**
	`Clang-LLVM`	314.61	314.23	10.74	269.61	272.17	4.63	**14.30%**
LLVM	`Clang-FFmpeg`	5314.00	5213.99	223.89	4630.73	4578.08	185.49	**12.86%**
	`Clang-LLVM`	5470.99	5401.73	208.07	4518.27	4453.56	157.49	**17.41%**
binutils	`Clang-FFmpeg`	180.17	180.53	3.53	149.71	149.60	1.82	**16.91%**
	`Clang-LLVM`	178.46	178.60	2.73	151.54	151.94	3.62	**15.08%**
xerces-c	`Clang-FFmpeg`	111.40	111.11	1.93	94.14	93.76	1.09	**15.49%**
	`Clang-LLVM`	112.00	113.70	2.67	92.27	91.75	1.30	**17.62%**

Table 1
Results from in between the stage 1 (Non-PGO) and stage 3 (PGO) analyses. We analyzed Non-PGO Clang against Clang-FFmpeg 5-5 times interchangeably, and Non-PGO Clang against Clang-LLVM similarly. In total, we analyzed each project 20 times. Each cell describes either the mean, median and standard deviation of 5 analyses.

5. Results and Evaluation

We show our results in Table 1. Our measurement confirms that the benefits of profile-guided optimization can be realized on the Clang Static Analyzer as well, not only on "regular" compilation. Analyses done with a PGO binary were 12.68-17.62% faster than the same analyses with a non-PGO binary.

In Table 2 we performed paired t-tests the project analyses. The tests shows with very high confidence that the realized speed improvements are statistically significant in all cases.

Project	Profile	N	Mean Diff (s)	SE Diff (s)	t-statistic	p-value
FFmpeg	`Clang-FFmpeg`	5	48.84	4.08	11.96	0.000280
	`Clang-LLVM`	5	45.00	6.58	6.84	0.002386
LLVM	`Clang-FFmpeg`	5	683.27	19.64	34.80	0.000004
	`Clang-LLVM`	5	952.72	26.44	36.03	0.000004
binutils	`Clang-FFmpeg`	5	30.46	1.44	21.09	0.000030
	`Clang-LLVM`	5	26.92	1.49	18.10	0.000055
xerces-c	`Clang-FFmpeg`	5	17.26	0.79	21.89	0.000026
	`Clang-LLVM`	5	19.73	1.16	16.99	0.000070

Table 2
Paired t-tests on our measurement data, indicating that the execution time advantage gained by profile-guided optimization is statistically significant.

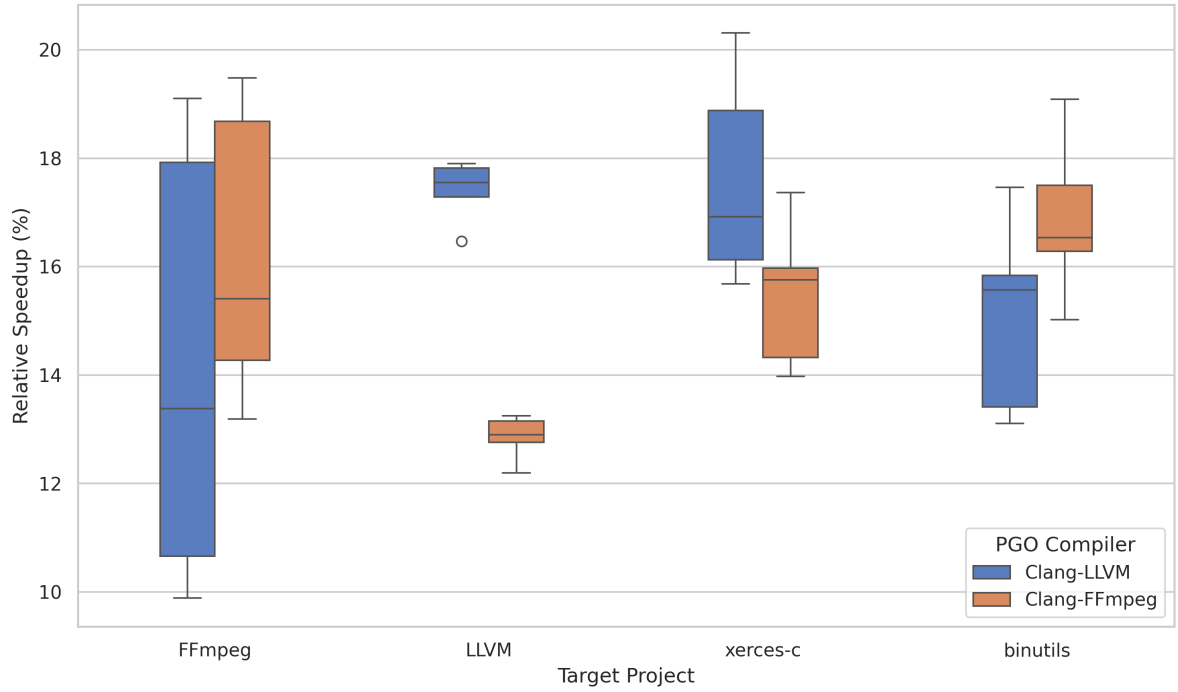


Figure 1: Paired box plots of our measurement data. Each box shows the spread of speedup against the non-PGO optimized Clang Static Analyzer.

Our measurement suggests that both Clang-FFmpeg and Clang-LLVM vastly outperform stage 1 compilers regardless on the project they analyzed, even though the former was profile-optimized on a pure C codebase, whereas the latter was optimized for a modern C++ codebase.

The lowest speedup figure at 12.68% was in between Non-PGO Clang and Clang-FFmpeg on the LLVM+Clang analysis. LLVM+Clang also showed the greatest speedup of 17.41% from Non-PGO Clang to Clang-LLVM. Interestingly, the difference in the speedup figures is not as stark on the analysis of FFmpeg: Clang-FFmpeg was 16.24% faster compared to Clang-LLVM's 14.30%.

A plausible explanation is how PGO actually works: it uses edge and path profiling to identify hot and cold branches, contextual execution correlations, and other data about the program, which is used to enable optimizations such as code-layout adjustment and informed inlining decisions [13]. As C is a subset of the C++ language, a profile compiled by performing static analysis on a C++ program also, in

part, profiles for C static analysis as well.

Nonetheless, while there our project set is limited, the results hint that the C projects (binutils, FFmpeg) were analyzed faster by Clang-FFmpeg, and the analyses on C++ projects (LLVM+Clang, xerces-c) were faster with Clang-LLVM. In Figure 1 we show paired boxplots of relative speedup of analysis time by Clang-LLVM and Clang-FFmpeg. This suggests that it might be worthwhile for static analysis distributors to ship both a C and a C++ optimized binary to efficiently analyze projects that combine the two language, such as LLVM itself.

6. Conclusion

As software takes a larger and larger part of everyday life, the detection of programming errors must be on the forefront as well. Automatic detection, and in particular, static analysis plays a crucial rule in this field. Symbolic execution, a prevalent static analaysis technique takes a radical approach to this: it essentially interprets the program code, finds deep rooted programming errors on entire classes of input values, but at the cost of high resource consumption.

Tools like the Clang Static Analyzer, which uses symbolic execution can take many hours to finish analysis. Alongside algorithmic optimizations, compiler optimizations must be used to decrease analysis time as much as possible. Among other optimization strategies, profile-guided optimization is a powerful technique to drastically improve execution performance.

In this paper, we investigated how to create a static analysis based profile, how to build an optimized Clang Static Analyzer with the help of this profile data, and we measured its effectiveness. We found that compared to non-profile-optimized binary, PGO reduced the analysis time by 12.68-17.62%.

As PGO depends on the profile data, we measured the impact of two different profiles: one gathered on the analysis of the C project FFmpeg, and one gathered on the C++ project LLVM. Our results hint that the C-optimized analyzer binary yields higher time savings on C projects, and similarly, the C++-optimized analyzer is faster on C++ projects.

Declaration on Generative AI

We used ChatGPT and Gemini for the purposes of Content Enhancement as defined in the CEUR GenAI Usage Taxonomy, primarily to gather literature and generate the programs used in evaluating our measurements. We reviewed and edited the tools' output as needed and take full responsibility for the publication's content. We did not use any generative AI for writing actual text of this paper.

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. Nemzeti Kutatási, Fejlesztési és Innovációs Hivatal

References

[1] 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.

[2] Synopsys, Coverity, 2019. https://scan.coverity.com/ (last accessed: 24-04-2023).

[3] Roguewave, Klocwork, 2019. https://www.roguewave.com/products-services/klocwork (last accessed: 28-02-2019).

[4] CodeSecure, CodeSonar, 2025. https://codesecure.com/our-products/codesonar/ (last accessed: 29-07-2025).
[5] 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.
[6] J. C. King, Symbolic execution and program testing, Commun. ACM 19 (1976) 385–394. URL: http://doi.acm.org/10.1145/360248.360252. doi:10.1145/360248.360252.
[7] 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%2Ficai.9.2014.2.309. doi:10.14794/icai.9.2014.2.309.
[8] 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.
[9] P. Boonstoppel, C. Cadar, D. Engler, RWset: Attacking Path Explosion in Constraint-Based Test Generation, Springer Berlin Heidelberg, 2008, p. 351–366. URL: http://dx.doi.org/10.1007/978-3-540-78800-3_27. doi:10.1007/978-3-540-78800-3_27.
[10] D. Schwartz-Narbonne, M. Schäf, D. Jovanović, P. Rümmer, T. Wies, Conflict-Directed Graph Coverage, Springer International Publishing, 2015, p. 327–342. URL: http://dx.doi.org/10.1007/978-3-319-17524-9_23. doi:10.1007/978-3-319-17524-9_23.
[11] V. Kuznetsov, J. Kinder, S. Bucur, G. Candea, Efficient state merging in symbolic execution, in: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 193–204. URL: https://doi.org/10.1145/2254064.2254088. doi:10.1145/2254064.2254088.
[12] Fisher, Trace scheduling: A technique for global microcode compaction, IEEE Transactions on Computers C-30 (1981) 478–490. URL: http://dx.doi.org/10.1109/tc.1981.1675827. doi:10.1109/tc.1981.1675827.
[13] B. Liu, Y. Huang, J. Gao, J. Shi, Y. Liu, Y. Sun, W. Ji, From profiling to optimization: Unveiling the profile guided optimization, arXiv (2025). doi:10.48550/arxiv.2507.16649.
[14] P. P. Chang, S. A. Mahlke, W. W. Hwu, Using profile information to assist classic code optimizations, Software: Practice and Experience 21 (1991) 1301–1321. URL: http://dx.doi.org/10.1002/spe.4380211204. doi:10.1002/spe.4380211204.
[15] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, ACM Comput. Surv. 51 (2018). doi:10.1145/3182657.
[16] F. S. de Boer, M. M. Bonsangue, On the nature of symbolic execution, in: 2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE, 2019, p. 4–5. URL: http://dx.doi.org/10.1109/synasc49474.2019.00009. doi:10.1109/synasc49474.2019.00009.
[17] Clang SA, Clang Static Analyzer, 2019. https://clang-analyzer.llvm.org/.
[18] Checker Developer Manual, Clang Static Analyzer: Checker Developer Manual, 2019. https://clang-analyzer.llvm.org/checker_dev_manual.html (last accessed: 24-04-2023).
[19] A. Dergachev, Clang Static Analyzer: A Checker Developer's Guide, 2016. https://github.com/haoNoQ/clang-analyzer-guide (last accessed: 24-04-2023).
[20] A. Zaks, J. Rose, Building a checker in 24 hours, 2012. https://www.youtube.com/watch?v=kdxlsP5QVPw.
[21] Á. Balogh, R. Szalay, On the applicability of static analysis for system software using codechecker, in: 2024 7th International Conference on Software and System Engineering (ICoSSE), 2024, pp. 15–22. doi:10.1109/ICoSSE62619.2024.00011.
[22] G. Horvath, R. Kovacs, R. Szalay, Z. Porkolab, G. Orban, D. Krupp, Static code analysis with codechecker, 2024. URL: https://arxiv.org/abs/2408.02220. arXiv:2408.02220.
[23] C. Lattner, Llvm and clang: Next generation compiler technology, 2008. Lecture at BSD Conference 2008.
[24] K. Umann, Z. Porkoláb, Detecting uninitialized variables in c++ with the clang static analyzer, Acta

Cybernetica 25 (2020) 923–940. URL: http://dx.doi.org/10.14232/actacyb.282900. doi:10.14232/actacyb.282900.

[25] G. Tóthvári, K. Umann, Z. Porkoláb, Finding bugs on tagged unions mismatches, in: 2024 IEEE 17th International Scientific Conference on Informatics (Informatics), 2024, pp. 415–420. doi:10.1109/Informatics62280.2024.10900816.

[26] G. Spaits, K. Umann, Z. Porkoláb, A survey of tagged union type usage in modern c++ projects, in: 2024 7th International Conference on Software and System Engineering (ICoSSE), 2024, pp. 31–36. doi:10.1109/ICoSSE62619.2024.00013.

[27] R. Kovács, G. Horváth, Z. Porkoláb, Detecting c++ lifetime errors with symbolic execution, in: Proceedings of the 9th Balkan Conference on Informatics, 2019, pp. 1–6.

[28] M. Zhou, B. Wu, X. Shen, Y. Gao, G. Yiu, Examining and reducing the influence of sampling errors on feedback-driven optimizations, ACM Transactions on Architecture and Code Optimization 13 (2016) 1–24. URL: http://dx.doi.org/10.1145/2851502. doi:10.1145/2851502.

[29] C. Zilles, G. Sohi, Execution-based prediction using speculative slices, in: Proceedings 28th Annual International Symposium on Computer Architecture, ISCA-01, IEEE Comput. Soc, 2001. URL: http://dx.doi.org/10.1109/isca.2001.937426. doi:10.1109/isca.2001.937426.

[30] T. Ball, J. R. Larus, Optimally profiling and tracing programs, in: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '92, POPL '92, ACM Press, 1992, p. 59–70. URL: http://dx.doi.org/10.1145/143165.143180. doi:10.1145/143165.143180.

[31] T. Ball, J. R. Larus, Efficient path profiling, in: Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29, MICRO-96, IEEE Comput. Soc. Press, 2009, p. 46–57. URL: http://dx.doi.org/10.1109/micro.1996.566449. doi:10.1109/micro.1996.566449.

[32] The LLVM Project, How to build clang and llvm with profile-guided optimizations, https://llvm.org/docs/HowToBuildWithPGO.html, 2026. Accessed: 2026-07-01.

[33] Free Software Foundation, GNU Binutils, https://sourceware.org/binutils/, 2026. Accessed: 2026-07-01.

[34] V. Subhash, Formats and codecs, in: Quick Start Guide to FFmpeg, Apress, 2023, pp. 17–21. URL: https://doi.org/10.1007%2F978-1-4842-8701-9_3. doi:10.1007/978-1-4842-8701-9_3.

[35] C. Lattner, V. Adve, Llvm: A compilation framework for lifelong program analysis & transformation (2004) 75.

[36] Apache Software Foundation, Apache xerces, 2022. URL: https://xerces.apache.org/.