Proceedings of the
13th International Conference on Applied Informatics
Eger, Hungary, February 19–21, 2026
DOI: 10.17048/icai.2026.96
pp. 96–106

Static Analysis Possibilities for Regular Expressions in C++

Tibor Teodor Fürész, Kristóf Umann, Zoltán Porkoláb

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

Abstract. Most of the regular expression libraries in current programming languages utilize a dynamic constructor parameter to construct a state machine, which will be used later in the matching. When this constructor parameter does not fulfill the requirements of the correct regular expression syntax, a runtime error – usually an exception – occurs. It would be highly beneficial if we could catch this kind of problem at compile time with the help of static analysis. The prerequisite of such an analysis is that the constructor parameter is fully known at compile time. In this research, we investigate the extent to which this prerequisite holds true for open-source software projects. We measured 26 different popular open-source projects, and we found that in more than two-thirds of the time, constructors used hardcoded parameters. This result opens the possibility for the creation of an effective static analysis tool.

Keywords: static analysis, C, C++, Clang, LLVM, regular expressions

1. Introduction

Regular expressions are an everyday part of software programming. Common scenarios like input parsing, file lookup, and search and replace are just a few examples, and in truth, it is hard to fathom a program with any meaningful complexity that can omit them. Most tools, like grep, that build regular expressions also come with robust techniques to check for their correctness.

Most of the current libraries work on the idea that regular expressions are implemented using state machines. The state machine is usually built up in the constructor, based on a regular expression pattern received as the construction parameter.

While this is a time-consuming process, later repeated use of the state machine is relatively cheap. However, the constructor parameter should be matched for the correct syntax of the regular expression. This error checking happens dynamically inside the constructor during the construction of the state machine.

During development, regular expressions are notoriously hard to get right. Their syntax is unintuitive, hard to read, and more importantly, to write [28]. Testing the expressions is also a challenge, as test effectiveness heavily depends on the test coverage. It is easy to see that test frameworks usually miss the same corner cases, which programmers also likely forget to implement correctly.

The trade-offs between dynamic and static analysis have been widely studied and publicized [4, 18, 21, 22, 30]. Briefly, most forms of dynamic analysis require the execution of the program and also precise input parameters. On the other hand, static analyzers usually do not require the concrete execution of the program or any input parameters, offering a unique advantage for detecting programming errors in the early stages of development [3]. As bugs that are caught early are faster, easier, and cheaper to fix [7], there are clear advantages for pursuing static analysis whenever possible.

This paper is organized as follows. We overview the related literature in Section 2. The technical background of our contribution is discussed in Section 3. Our measurement methodology will be introduced in Section 4. We evaluate our results in Section 5. Future research directions are discussed in Section 6. Finally, our paper concludes in Section 7.

2. Related work

Our paper investigates how C++ programmers use regular expressions, especially whether constructors use hard-wired string literals or constants. We found similar studies to ours aiming at how developers use certain C++ features. The authors in [19] survey the usage of immutability declarations. In a broader study, the authors in [48] surveyed how C++ libraries are used.

In their paper, the authors of [15] surveyed library usage of Fedora Linux 37, which spans just shy of 400 million lines of code. Among other interesting findings, they identified Boost as the most common third party library, and that the most frequently used assets from Boost have no, or only partial counterparts in the C++ standard library. They found that there is a *"worrying trend"* that programmers do not use bounds checking alternatives to access containers or views.

Regular expression implementations that have a super-linear worst-case complexity are vulnerability that can be exploited for denial of service attacks [13].

There are several varieties and extensions to regular expressions implemented in programming languages. In 2019, a study [14] discussed whether regular expressions written in one language can be ported as-is from one language to another. Across 193,524 projects and 537,806 regexes they found if ported, 15% posed a risk logic errors due to semantic differences, 10% might have performance differences to the extent where they posed a security risk.

Vulnerabilities connected to regular expressions and their usage are long time in the focus of research. One critical problem regarding regular expressions is the quality of their tests. Regular expressions are often less thoroughly tested than other parts of the codebase, according to a 2016 survey [8] with python developers. In 2018, a paper [46] measured that only 17% of regexes are tested in the 1225 Java projects they inspected, and even among those, their coverage (positive and negative test strings) were poor. The *AREXT* [44] tool is developed to automatically find, and generate test strings against regular expressions for full coverage, inspired by the former two papers.

The authors of paper [25] describe how "evil" test strings can be generated for testing regular expressions.

The *Boost.Xpressive* library provides functions to search strings using regular expressions [32]. Boost.Xpressive makes it possible to write down regular expressions as C++ code rather than strings. That makes it possible to check at compile time whether a regular expression is valid or not. However, Boost.Xpressive heavily utilizes C++ template metaprogramming [1, 45], which makes its syntax very different from other regular expression libraries.

While not specific to regular expressions, prior work highlights broader efforts to improve reasoning about program properties. For example, ARC (Automated Reasoning in the Class) proposes the deployment of automated reasoning tools in education, and surveys of specification languages categorize how properties can be expressed and checked (statically or dynamically), providing context for specialized analysis tools, such as ours [16, 24].

3. Technical overview

In the current C++ development, the landscape of regular expressions is fairly fragmented. Currently, there are three major libraries providing regex capabilities. These are the standard library (`std::regex`), Boost.Regex and Google's RE2. Each of them uses a different grammar by default. The standard library uses the ECMAScript 262 standard, which is an older version of the standard used in JavaScript. The Boost.Regex library defaults to the PERL regex grammar, which is the most feature-rich of the three. Finally, Google's RE2 defines its own grammar. The RE2 regex library promises linear matching time for every pattern. To achieve this developers had to strip functionality present in most other regex libraries, such as backreferences. It further complicates things that most libraries support multiple standards. For instance, the standard and boost regex both support the POSIX Basic Regular Expression (BRE) grammar, which is the default in grep, as well as the POSIX Extended Regular Expression (ERE) grammar, used in awk.

In the midst of these mountains of standards, developers are highly susceptible to making mistakes when writing their patterns. In Figure 1 we show a few possible errors regarding the standard `std::regex` class. One frequent mistake is trying to use a construct that is not supported by the currently selected standard. Other common syntax errors can originate from structural mistakes within the pattern.

These include not closing capture groups or exact quantifiers and improperly adding quantifiers after non-quantifiable tokens. Another hard-to-detect mistake is invalid escape sequences. Such a mistake can easily be made if the developer forgets to use a raw string for their patterns.

```
std::regex re("(?<=CodeChecker )clang-tidy"); // Using construct not
    supported by the standard
std::regex re("(abc"); // unclosed capture group
std::regex re("a{2");  // unclosed exact quantifier
std::regex re("a?*");  // quantifier after a non-quantifiable token
std::regex re("\d+");  // invalid escape sequence, matches "d", not
    digits
```

Figure 1. Example of possible syntax errors in regex constructors.

4. Measurement methodology

To conduct our measurements on C++ projects, we developed a custom static analysis checker built on the LLVM/Clang infrastructure. Specifically, we extended clang-tidy 22.0.0, a widely adopted tool, by implementing a custom checker to identify the instantiation of regex objects. The primary goal of our addition consists of two parts. First is to measure a baseline of the usage of regular expressions, and second is to determine in each case whether the pattern is known at compile time or not.

To achieve our first goal, we made our check emit a warning for each constructor call, regardless of what parameters it received. This unconditional matching allows us to safely count the number of times a regex is used in a project.

To complete our secondary goal, we perform a deeper analysis of the parameters received by the constructor. We try to classify each parameter into one of three categories. The first category is hardcoded patterns. These are patterns that are known at compile time. We regard each parameter as such if they are a string literal, or a constant string or constant character pointer defined with a single string literal. Our second category consists of patterns, which we determined to be constructed during runtime. We regard a parameter to be such a dynamic pattern if there is a concatenation operator in either the parameter itself or in the definition of the parameter variable. The final category is the unknown patterns, consisting of parameters that do not fall into either of the previous categories.

To account for a wide variety of programming standards and conventions, we took measurements in a wide selection of 26 open-source projects. These projects were sourced from GitHub with the only criteria being that they have at least one regex constructor in their codebase.

5. Results and evaluation

The summary of our findings is presented in Table 1. We can see a dominant usage of hardcoded regex patterns over dynamically constructed ones among the tested projects. Evaluating the results on a per-project basis, as shown in Table 2, revealed that in a significant portion of analyzed projects, every single regex pattern is hardcoded.

The results also show that only in a small number of cases can we determine that the pattern is constructed during runtime, meaning that with improvements made to our static analysis tool, we might be able to find even more hardcoded patterns. These results support our theory that there is a possibility to create a static analysis tool targeting regexes in C++.

Table 1. Overall number of regex constructors found in open-source projects.

Constructors	462	–
Constructors with hardcoded strings	322	69.70%
Constructors with dynamic strings	13	2.81%
Constructors unknown patterns	127	27.49%

A detailed investigation can reveal certain static and dynamic patterns from different libraries. A typical hardcoded regex constructor parameter can be seen in Figure 2. Bear uses it to escape all characters not defined in its group and wrap new lines into strings. An example of a dynamic pattern can be seen in Figure 3. SQLCheck uses a dynamic pattern to search for recursive references.

```
std::string escape(const std::string& input) {
  if (input.empty()) {
    return "''";
  }

  const std::regex ESCAPE_PATTERN(R"#(([^A-Za-z0-9_\-.,:/@\n]))#");
  const std::regex LINE_FEED(R"#(\n)#");

  const auto output =
      std::regex_replace(input, ESCAPE_PATTERN, "\\$1");
  return std::regex_replace(output, LINE_FEED, "'\n'");
}
```

Figure 2. Hardcoded patterns in Bear.

Table 2. Distribution of regular expression constructors and the type of their patterns.

Project	Total Regexes	hardcoded	Dynamic	Unknown
Bear [5]	10	10	0	0
Crow [12]	2	1	0	1
bitcoin [38]	2	1	0	1
blender [6]	23	20	2	1
ceph [39]	62	42	6	14
colmap [10]	6	6	0	0
contour [11]	6	6	0	0
drogon [17]	30	14	1	15
folly [27]	14	8	0	6
json [23]	1	1	0	0
libdatachannel [26]	2	2	0	0
llvm-project [40]	3	3	0	0
mold [29]	14	14	0	0
nanogui [31]	2	0	0	2
obs-studio [33]	4	2	0	2
pistache [34]	12	11	0	1
proxysql [35]	58	13	0	45
qt-6.10.1 [41]	33	27	0	6
retdec [2]	34	32	1	1
seastar [36]	12	4	0	8
shaderc [20]	24	11	2	11
sqlcheck [37]	35	32	1	2
threepp [43]	25	24	0	1
whisper.cpp [47]	17	11	0	6
xournalpp [49]	20	20	0	0
zeek [42]	11	7	0	4

6. Future work

In Figure 4 we can see a dynamic constructor parameter. This parameter is constructed with the concatenation of a number of namespace variables, therefore, we currently do not consider it as hardcoded. Although one can recognize that the variables themselves are hardcoded, and moreover, they are constant. An advanced data flow analysis could detect this fact and consider the constructor parameter as a hardcoded one. However, we then still have to model the string concatenation to obtain the final regex pattern and execute an appropriate syntactic check on it. This is one of our future research directions.

```
void CheckRecursiveDependency(Configuration& state,
                              const std::string& sql_statement,
                              bool& print_statement){

 std::string table_name = GetTableName(sql_statement);
 if(table_name.empty()){
   return;
 }

 std::regex pattern("(references\\s+" + table_name+ ")");
 std::string title = "Recursive Dependency";
 PatternType pattern_type = PatternType::
   PATTERN_TYPE_LOGICAL_DATABASE_DESIGN;

 auto message = "...";
 CheckPattern(...);

}
```

Figure 3. Dynamic patterns in SQLCheck.

```
...
const std::string schema_re = "([[:alpha:]]+:\\/\\/)";
const std::string user_pass_re = "(([^:\\s]+):([^@\\s]+)@)?";
const std::string host_port_re = "([[:alnum:].:-]+)";
const std::string path_re = "(/[[:print:]]*)?";
...

bool parse_url_authority(const std::string& url, std::string& host,
  std::string& user, std::string& password) {
  const std::string re = schema_re + user_pass_re + host_port_re +
  path_re;
  const std::regex url_regex(re, std::regex::icase);
  std::smatch url_match_result;

  if (std::regex_match(url, url_match_result, url_regex)) {
    host = url_match_result[HOST_GROUP_IDX];
    user = url_match_result[USER_GROUP_IDX];
    password = url_match_result[PASSWORD_GROUP_IDX];
    return true;
  }

  return false;
}
```

Figure 4. Quasi-dynamic patterns in Ceph.

7. Conclusion

This paper discusses the possibilities of static analysis of regular expressions with a particular focus on C/C++. Despite the age and the wealth of knowledge accumulated in the field of regular expressions, as of writing, we are not aware of any works that statically check the correctness of types like `std::regex` or `boost::regex`.

However, static analysis could be applied for regular expressions only if the expression is available at compile-time. Before any effort is made in this direction we have to understand how programmers use regular expressions, especially at what rate they apply patterns known at compilation time.

We applied the clang-tidy [9] static analysis tool to detect regular expression objects in large open-source C++ projects and to determine the ratio in which those objects were initialized with compile-time constant strings. We found that in typical C++ projects, most of the regular expression objects are initialized based on hardcoded string patterns, therefore static analysis of them is viable.

The paper describes the method, the algorithm, a prototype implementation, and the measurement results on large open-source projects. Our results proved that 69.7% of the regular expression objects are initialized with hardcoded string patterns, as seen in Table 1. Thus, this research could be the groundwork for the development of such a static analyzer.

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] D. Abrahams, A. Gurtovoy: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, C++ in Depth Series, Addison-Wesley Professional, 2004, isbn: 0321227255.

[2] Avast Software: *RetDec: A Retargetable Machine-Code Decompiler*, Source code repository, 2026, url: https://github.com/avast/retdec.

[3] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, J. Penix: *Using Static Analysis to Find Bugs*, IEEE Software 25.5 (Sept. 2008), pp. 22–29, doi: 10.1109/ms.2008.130, url: https://doi.org/10.1109%2Fms.2008.130.

[4] 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.

[5] Bear Contributors: *Bear: Build EAR*, Source code repository, 2026, url: https://github.com/rizsotto/Bear.

[6] Blender Foundation: *Blender*, Source code repository, 2026, URL: https://github.com/blender/blender.

[7] B. Boehm, V. R. Basili: *Software Defect Reduction Top 10 List*, Computer 34.1 (Jan. 2001), pp. 135–137, ISSN: 0018-9162, DOI: 10.1109/2.962984, URL: http://dx.doi.org/10.1109/2.962984.

[8] C. Chapman, K. T. Stolee: *Exploring regular expression usage and context in Python*, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA '16, ACM, July 2016, pp. 282–293, DOI: 10.1145/2931037.2931073, URL: http://dx.doi.org/10.1145/2931037.2931073.

[9] Clang Tidy: *Clang-Tidy*, https://clang.llvm.org/extra/clang-tidy/ (last accessed: 28-02-2019)., 2019.

[10] COLMAP Contributors: *COLMAP: A General-Purpose Structure-from-Motion and Multi-View Stereo Pipeline*, Source code repository, 2026, URL: https://github.com/colmap/colmap.

[11] Contour Contributors: *Contour: Modern C++ Terminal Emulator*, Source code repository, 2026, URL: https://github.com/contour-terminal/contour.

[12] CrowCpp Contributors: *Crow: A Fast C++ Microframework for the Web*, Source code repository, 2026, URL: https://github.com/CrowCpp/Crow.

[13] J. C. Davis, C. A. Coghlan, F. Servant, D. Lee: *The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale*, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE '18, ACM, Oct. 2018, pp. 246–256, DOI: 10.1145/3236024.3236027, URL: http://dx.doi.org/10.1145/3236024.3236027.

[14] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant, D. Lee: *Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions*, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE '19, ACM, Aug. 2019, pp. 443–454, DOI: 10.1145/3338906.3338909, URL: http://dx.doi.org/10.1145/3338906.3338909.

[15] J. Deng, M. D. Adams: *An Analysis of Library Usage in the C++ Code Base of Fedora Linux 37*, in: 2024 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), IEEE, Aug. 2024, pp. 1–6, DOI: 10.1109/pacrim61180.2024.10690215, URL: http://dx.doi.org/10.1109/pacrim61180.2024.10690215.

[16] I. Dramnesc, T. Jebelean, E. Abraham, G. Kusper, S. Stratulat: *ARC: An Educational Project on Automated Reasoning in the Class*, in: Proceedings of EdMedia 2022, ed. by T. Bastiaens, Online: Association for the Advancement of Computing in Education (AACE), Nov. 2022, pp. 934–943, URL: https://www.learntechlib.org/p/221395.

[17] Drogon Framework Contributors: *Drogon: A C++14/17/20 based HTTP Web Framework*, Source code repository, 2026, URL: https://github.com/drogonframework/drogon.

[18] M. D. Ernst: *Static and dynamic analysis: Synergy and duality*, in: WODA 2003: ICSE Workshop on Dynamic Analysis, 2003, pp. 24–27, DOI: 10.1109/icse.2003.1201290.

[19] J. Eyolfson, P. Lam: *How C++ Developers Use Immutability Declarations: An Empirical Study*, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 362–372, DOI: 10.1109/ICSE.2019.00050.

[20] Google LLC: *Shaderc: A collection of tools, libraries, and tests for Vulkan shader compilation*, Source code repository, 2026, URL: https://github.com/google/shaderc.

[21] S. Hallem, B. Chelf, Y. Xie, D. Engler: *A System and Language for Building System-specific, Static Analyses*, PLDI '02 (2002), pp. 69–82, DOI: 10.1145/512529.512539, URL: http://doi.acm.org/10.1145/512529.512539.

[22] S. Heckman, L. Williams: *A systematic literature review of actionable alert identification techniques for automated static code analysis*, Information and Software Technology 53.4 (2011), Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing, pp. 363–387, ISSN: 0950-5849, DOI: https://doi.org/10.1016/j.infsof.2010.12.007, URL: https://www.sciencedirect.com/science/article/pii/S0950584910002235.

[23] JSON Contributors: *JSON for Modern C++*, Source code repository, 2026, URL: https://github.com/nlohmann/json.

[24] G. Kusper, G. Kovásznai, W. Schreiner, G. Guta, J. Sztrik: *A Small Survey of Java Specification Languages*, in: Jan. 2010.

[25] E. Larson, A. Kirk: *Generating Evil Test Strings for Regular Expressions*, in: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, Apr. 2016, pp. 309–319, DOI: 10.1109/icst.2016.29, URL: http://dx.doi.org/10.1109/icst.2016.29.

[26] libdatachannel Contributors: *libdatachannel: C/C++ WebRTC network library*, Source code repository, 2026, URL: https://github.com/paullouisageneau/libdatachannel.

[27] Meta Platforms, Inc.: *Folly: Facebook Open-source Library*, Source code repository, 2026, URL: https://github.com/facebook/folly.

[28] L. G. Michael, J. Donohue, J. C. Davis, D. Lee, F. Servant: *Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions*, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 415–426, DOI: 10.1109/ASE.2019.00047.

[29] Mold Contributors: *Mold: A Modern Linker*, Source code repository, 2026, URL: https://github.com/rui314/mold.

[30] N. Nagappan, T. Ball: *Static Analysis Tools As Early Indicators of Pre-release Defect Density*, ICSE '05 (2005), pp. 580–586, DOI: 10.1145/1062455.1062558, URL: http://doi.acm.org/10.1145/1062455.1062558.

[31] NanoGUI Contributors: *NanoGUI: Minimalist GUI library for OpenGL*, Source code repository, 2026, URL: https://github.com/wjakob/nanogui.

[32] E. Niebler: *Boost.Xpressive: An advanced regular expression template library for C++*, https://boost.org, Boost C++ Libraries, 2007.

[33] OBS Project Contributors: *OBS Studio*, Source code repository, 2026, URL: https://github.com/obsproject/obs-studio.

[34] Pistache.io: *Pistache: An elegant C++ REST framework*, Source code repository, 2026, URL: https://github.com/pistacheio/pistache.

[35] ProxySQL Contributors: *ProxySQL: High Performance, High Availability, Protocol Aware Proxy for MySQL*, Source code repository, 2026, URL: https://github.com/sysown/proxysql.

[36] ScyllaDB: *Seastar: High performance server-side application framework*, Source code repository, 2026, URL: https://github.com/scylladb/seastar.

[37] SQLCheck Contributors: *SQLCheck: Automatically identify anti-patterns in SQL queries*, Source code repository, 2026, URL: https://github.com/jarulraj/sqlcheck.

[38] The Bitcoin Core: *Bitcoin Core*, 2022, URL: https://bitcoincore.org/.

[39] The Ceph Foundation: *Ceph: A distributed object, block, and file storage platform*, Source code repository, 2026, URL: https://github.com/ceph/ceph.

[40] The LLVM Project: *The LLVM Compiler Infrastructure*, Source code repository, 2026, URL: https://github.com/llvm/llvm-project.

[41] The Qt Company: *Qt 6*, Source Code Archive, 2026, URL: https://download.qt.io/archive/qt/6.10/6.10.1/single/.

[42] The Zeek Project: *Zeek Network Security Monitor*, Source code repository, 2026, URL: https://github.com/zeek/zeek.

[43] threepp Contributors: *threepp: C++ port of three.js*, Source code repository, 2026, URL: https://github.com/markaren/threepp.

[44] N. Van Hoan, P. N. Hung: *Arext: Automatic Regular Expression Testing Tool Based on Generating Strings With Full Coverage*, in: 2021 13th International Conference on Knowledge and Systems Engineering (KSE), IEEE, Nov. 2021, pp. 1–6, DOI: 10.1109/kse53942.2021.9648604, URL: http://dx.doi.org/10.1109/kse53942.2021.9648604.

[45] D. van de Voorde, N. M. Josuttis, D. Gregor: *C++ Templates: The Complete Guide (2nd Edition)*, 2nd, Addison-Wesley Professional, Sept. 2017, ISBN: 978-0-321-71412-1, URL: http://tmplbook.com.

[46] P. Wang, K. T. Stolee: *How well are regular expressions tested in the wild?*, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE '18, ACM, Oct. 2018, pp. 668–678, DOI: 10.1145/3236024.3236072, URL: http://dx.doi.org/10.1145/3236024.3236072.

[47] Whisper.cpp Contributors: *Whisper.cpp: High-performance inference of OpenAI's Whisper automatic speech recognition*, Source code repository, 2026, URL: https://github.com/ggml-org/whisper.cpp.

[48] D. Wu, L. Chen, Y. Zhou, B. Xu: *How do developers use C++ libraries? An empirical study*, in: Proceedings of the Twenty-Seventh International Conference on Software Engineering and Knowledge Engineering, 2015, pp. 260–265.

[49] Xournal++ Contributors: *Xournal++: Handwriting notetaking software with PDF annotation*, Source code repository, 2026, URL: https://github.com/xournalpp/xournalpp.