JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 35 (2025) 216–223
Received: 06 March 2025
Accepted: 23 April 2025

Security vulnerabilities in C++ programs

Luki w bezpieczeństwie programów w języku C++

Piotr Michał Adamczyk*, Marek Miłosz

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

Software security is a challenge posed to modern programming developers it is important not only to protect data and resources, but also to ensure stability, reliability and confidence in the systems used. The C++ language, due to its lack of memory control and high flexibility, is particularly prone to security vulnerabilities. The aim of this paper is to review the literature to evaluate the effectiveness of existing methods to detect and prevent security vulnerabilities in programmes written in C++. The literature analysis showed that static analysis tools are effective in detecting most common vulnerabilities. However, they need to be combined with other methods to eliminate most security vulnerabilities.

Keywords: Security vulnerabilities; C++; Software security analysis

Streszczenie

Bezpieczeństwo oprogramowania jest wyzwaniem postawionym współczesnym twórcom programowania ma ono ważne znaczenie nie tylko dla ochrony danych i zasobów, ale także dla zapewnienia stabilności, niezawodności i zaufania do używanych systemów. Język C++ ze względu na brak kontroli pamięci oraz wysoką elastyczność jest szczególnie narażony na występowanie luk bezpieczeństwa. Celem niniejszego artykułu jest przegląd literatury w celu oceny skuteczności istniejących metod do wykrywania oraz zapobiegania lukom bezpieczeństwa w programach napisanych w języku C++. Analiza literatury wykazała, że narzędzia do analizy statycznej są skuteczne w wykrywaniu większości typowych podatności. Jednak, aby wyeliminować większość luk bezpieczeństwa należy je połączyć z innymi metodami.

Słowa kluczowe: Luki w bezpieczeństwie; C++; Analiza bezpieczeństwa oprogramowania

*Corresponding author

Email address: piotr.adamczyk@pollub.edu.pl (P. M. Adamczyk)

1. Introduction

The purpose of this article is to conduct a systematic literature review to systematize the current state of knowledge about the effectiveness of existing methods and tools used to detect and prevent security vulnerabilities in programs written in C++.

The C++ language is characterized by high complexity and rich functionality, which makes programmers often make mistakes. Code complexity increases as new features are added, increasing the risk of errors and security vulnerabilities [1].

C++ gives programmers a great deal of control over memory management, which can lead to errors such as memory leaks, invalid memory access (Invalid Memory Access), and buffer overflow (Buffer Overflow) errors. Improper use of pointers is particularly problematic and often leads to serious security vulnerabilities [2].

Many security vulnerabilities result from a lack of proper validation of input data. Unverified data can lead to attacks such as buffer overflows which allows attackers to execute arbitrary code on the attacked system [3].

2. Purpose and scope of work

The purpose of this paper is to conduct a systematic literature review of methods and tools used to identify and prevent security vulnerabilities in software developed in C++. The paper focuses on an analysis of the current state of the art, covering both the theoretical foundations and practical aspects related to security in C++ programming.

The main objective is to evaluate the effectiveness of existing methods and tools used in security analysis, with a particular focus on their ability to detect vulnerabilities such as buffer overflow, null pointer dereferences or use-after-free. As part of this evaluation, an important element of the work will be to answer the research question: Are static code analysis tools more effective than others in detecting security vulnerabilities in programs written in C++ compared to other methods such as dynamic analysis or security testing?

The article also aims to make recommendations for selecting the best tools and techniques to support the process of ensuring software security in C++. These recommendations, based on the results of the literature review, will be aimed at programmers and development teams and are intended to facilitate informed decisions on tools to support the development of secure code in C++.

The review work is limited to the analysis of available research and literature and does not include conducting experiments or testing practical tools. Its results are intended to provide a structured overview of the current state of knowledge in the area of software security in C++.

3. Literature review

C++ is a programming language developed by Danish computer scientist Bjarne Stroustrup. It was designed as an improvement of the C language with object-oriented programming, better type checking and data abstractions [4]. The first version was 1985 but is still being developed today by the organisation 'Standard C++ Foundation', which is currently working on the next version of the language, designated C++26 [5,6].

C++ is known for certain types of security vulnerabilities due to its low-level capabilities and manual memory management. The most common security vulnerability in C++ is Buffer Overflow [7]. The main categories of these vulnerabilities are described below:

1. Buffer Overflow - a buffer overflow occurs when more data is written to a buffer than it can hold, leading to the overwriting of adjacent memory cells. This can lead to unpredictable behaviour, crashing or being used by attackers to execute arbitrary code. Buffer overflows are a critical security vulnerability, especially in programs written in C++ due to the lack of a built-in function to check the boundaries of buffer operations [2,8].
2. Memory Leaks - a memory leak occurs when a computer program mismanages memory allocation, leading to a situation where memory that is no longer needed is not released. This can result in a gradual decrease in available memory, which can reduce system performance or cause the program to crash [9].
3. Invalid Memory Access - invalid memory access occurs when a program attempts to read from or write to a memory cell for which it has no authorization. This often causes crashes or unpredictable behaviour, such as segmentation errors on Unix or Linux systems. This is a common bug for C++ programs that manage memory manually without built-in security controls [10].
4. Unvalidated Input - unvalidated input refers to scenarios where an application receives data from an external source (user, network, file, etc.) and does not properly check or sanitize that data before using it. This can lead to numerous security vulnerabilities, as the application may inadvertently run malicious data or allow it to affect its operation [3].
5. Integer Overflow - integer overflow occurs when an arithmetic operation attempts to create a numeric value that exceeds the range represented by the number's data type. This can happen during addition, subtraction, multiplication, or casting between different integer types (e.g., from an integer with sign to an integer without sign) [11].

Vulnerability prevention techniques - preventing vulnerabilities in code written in C++ is a key aspect of secure programming. There are many methods and tools that can help minimize the risk of vulnerabilities:

1. Static analysis - this involves reviewing source code without running it to detect potential bugs and security vulnerabilities. Static tools analyse code for known error patterns, such as unprotected pointers, buffer overflows and memory management errors [12].
2. Dynamic analysis - this involves running a program and observing its behaviour under real-world conditions. This allows detection of errors that only occur in real time, such as memory leaks or invalid memory accesses. Dynamic tools monitor resource usage and program behaviour to identify anomalies [13].
3. Validation and sanitization of input data - a fundamental technique for preventing gaps. Any data that comes in from the user must be carefully checked to ensure that it conforms to expected formats and value ranges. Techniques such as regular expressions, input length checking and format checking can prevent Buffer Overflow attacks [14].
4. Proper memory management is crucial in C++ to avoid memory leaks and corruption. It is advisable to use smart pointers and the RAII (Resource Acquisition Is Initialization) technique which automates resource management and minimizes the risk of memory leaks [14].
5. Code testing - automated code testing tools, and manual code reviews can help detect potential security vulnerabilities prior to application deployment. Application testing, including penetration testing, also plays a key role in identifying and fixing vulnerabilities [15].
6. Fuzzing - this is a technique for testing software, detecting bugs, unexpected behaviour and security vulnerabilities. It works by continuously generating test cases and verifying the state of the program [16].

4. Research methods

The methodology used in this study is based on the approach presented in the work of Skublewska-Paszkowska et al. [17] and includes three main stages: Preparation, Implementation of the search process, and Data Synthesis. This process provides a systematic and reliable approach to the analysis of scientific literature, enabling the evaluation of the effectiveness of methods and tools related to the detection and prevention of security vulnerabilities in C++. A diagram of this methodology is shown in Figure 1.

- Stage 1: Preparation - provides the foundation for the entire research process. Its purpose is to lay a solid foundation that enables a systematic and focused approach to the literature review. The stage consists of three steps: formulating research questions, developing search queries, and selecting bibliographic databases.
    - Formulating research questions - involves defining questions that address issues related to the research objectives. Thanks to these questions, the study gains an orderly structure of the scope of the collected, which makes it possible to systematically analyse the literature.
    - Developing search queries - is the identification of key phrases or terms that best reflect the subject matter of the study. Queries should be constructed using logical operators (e.g., AND, OR) to precisely narrow down the search results. This ensured that the process of searching the selected databases was systematic and repeatable.

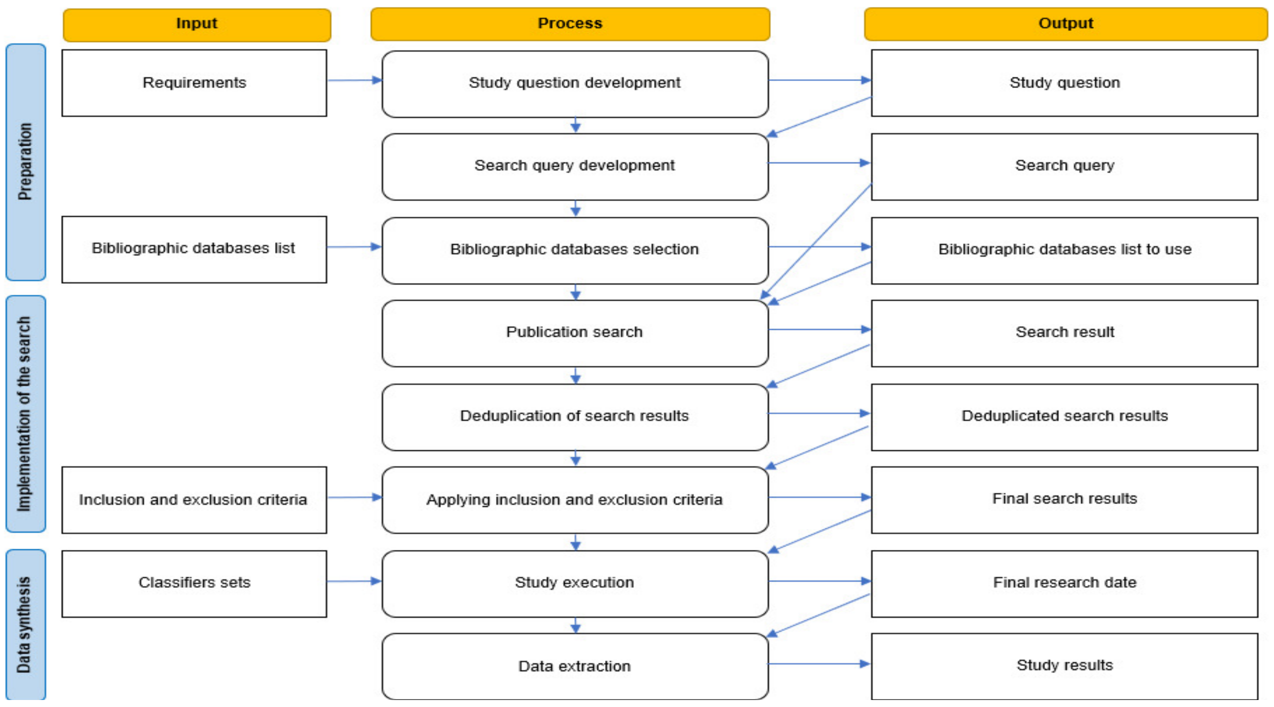
Input
Process
Output
Preparation
Implementation of the search
Data synthesis
Requirements
Bibliographic databases list
Inclusion and exclusion criteria
Classifiers sets
Study question development
Search query development
Bibliographic databases selection
Publication search
Deduplication of search results
Applying inclusion and exclusion criteria
Study execution
Data extraction
Study question
Search query
Bibliographic databases list to use
Search result
Deduplicated search results
Final search results
Final research date
Study results

Figure 1: Study methodology [17].

  - Selection of bibliographic databases - the selection of appropriate bibliographic databases allows access to high-quality scientific literature and to reach up-to-date and reliable materials, corresponding to the topic of the work.
- Stage 2: Implementing the search process - the goal of this stage is to collect relevant source materials that corresponded to the topic of the study, and to preliminarily analyse them to eliminate redundant or inadequate content. The stage includes three steps: publication search, removal of duplicates, and application of inclusion and exclusion criteria.
  - Publication search - involves searching selected bibliographic databases using previously developed search queries.
  - Removal of duplicates - the elimination of duplicate results obtained from different databases ensures that the same publications are not viewed multiple times, which improves further analysis and organization of results.
  - Application of inclusion and exclusion criteria - publications should be evaluated for their compatibility with the topic of the paper, whether they contain research relevant to the topic, or other criteria established in advance.
- Stage 3: Data synthesis - involves detailed analysis and extraction of key information from the retrieved publications, organizing them, and formulating conclusions corresponding to the research questions. The stage is divided into survey execution and data extraction.
  - Execution of the survey - conducting a detailed analysis of the content of each selected publication and giving them classifiers corresponding to the content of the research questions.
  - Data extraction - analysing the collected data after performing the survey.

Search query - the query “security vulnerabilities” AND “C++ programs” was constructed to search for publications on security vulnerabilities in programs written in C++.

The keyword “security vulnerabilities” focuses on the general concept of security vulnerabilities, which includes all vulnerabilities, bugs and problems related to software security and “C++ programs” narrows the results to publications that directly relate to programs written in C++, allowing the search to focus on C++ specific issues. Using the logical operator AND means that the searched publications must contain both phrases at the same time. This will ensure that the results specifically address security vulnerabilities in the context of programs in C++.

Bibliographic databases - the authors were able to use the following bibliographic databases: AccessEngineering, AIP, APS, BIBLIO Ebookpoint, Building Types Online, EBSCO eBook, Emerald, EMIS Intelligence, IBUK Libra, IEEE Xplore, JCR, JSTOR, MathSciNet, Medline, Nature, Science, ScienceDirect, Scopus, SpringerLink, TotalMateria, Trans Tech Publications, Web of Science, Wiley Online Library. From the above-mentioned databases, the most reputable in the field of technology were selected: Scopus, SpringerLink, Web of Science and IEEE Xplorer.

Selection of classifiers - the study uses three classifiers to systematize and analyse the selected publications.

1. Year - allows identifying trends in C++ security research and the evolution of tools and methods for detecting security vulnerabilities.
2. Research type - publications were divided based on their type: Conference Proceedings, Journal Article, Book.

3. Category - depending on the methods and tools described in the publication, they were assigned to categories: Fuzzing, Static analysis, Dynamic analysis, Hardware support, Tests. This classifier allowed to evaluate the effectiveness of a specific category in solving security problems.

5. Study results

The results of the literature review are presented in Table 1. It shows the breakdown of the analysed publications [18–55] by year of publication, type of study and category.

Table 1: Results from the review

#	Author(s)	Year	Research type	Fuzzing	Static	Dynamic	Hardware	Tests
1	Blackwell et al.	2025	Journal Article	X				
2	Song et al.	2024	Conference Proceedings		X			
3	Seas et al.	2024	Conference Proceedings		X			
4	Alshmrany et al.	2024	Journal Article	X	X			
5	Kasten et al.	2024	Conference Proceedings		X			
6	Kaivo et al.	2023	Conference Proceedings			X		
7	Liu et al.	2023	Conference Proceedings		X			
8	Park et al.	2023	Conference Proceedings				X	
9	Shen et al.	2023	Journal Article		X			
10	Hohentanner et al.	2023	Conference Proceedings		X		X	
11	Alshmrany et al.	2022	Journal Article	X	X			
12	Godboley et al.	2022	Conference Proceedings	X	X			
13	Monteiro et al.	2022	Journal Article		X			
14	Li et al.	2022	Journal Article		X			
15	Machiry et al.	2022	Journal Article		X			
16	Alshmrany et al.	2021	Journal Article	X	X			
17	Zaharia et al.	2021	Conference Proceedings		X			
18	Alshmrany et al.	2021	Conference Proceedings	X	X			
19	Iqbal et al.	2021	Conference Proceedings		X			
20	Gao et al.	2020	Journal Article		X	X		
21	Rong et al.	2020	Conference Proceedings	X				
22	Giet et al.	2019	Journal Article		X			
23	Zou et al.	2019	Conference Proceedings		X			
24	Liu et al.	2019	Conference Proceedings	X				
25	Gu et al.	2019	Conference Proceedings		X			
26	Lu et al.	2018	Journal Article		X			
27	Hermeling	2018	Journal Article		X			
28	Le et al.	2018	Journal Article		X			
29	Gerasimov et al.	2018	Journal Article		X	X		
30	Biondi et al.	2018	Journal Article		X			
31	Dimjašević et al.	2018	Journal Article		X	X		X
32	Maurica et al.	2018	Conference Proceedings		X	X		X
33	Gerasimov	2018	Journal Article		X	X		
34	Ye et al.	2018	Conference Proceedings		X		X	
35	Bican et al.	2018	Conference Proceedings		X			
36	Vorobyov et al.	2018	Conference Proceedings			X		
37	Cook et al.	2018	Journal Article		X			
38	Chen et al.	2018	Journal Article		X	X		X

The results of the number of publications collected from various databases are presented in Table 2, which shows the number of articles obtained from Scopus, SpringerLink, Web of Science and IEEE Xplorer sources.

Table 2: Analysed papers in the following databases

	Searched papers	Excluded papers	Included papers
Scopus	65	50	15
SpringerLink	201	181	20
Web of Science	36	25	11
IEEE Xplorer	31	20	11

Figure 2 presents an overview of categories assigned to publications: Fuzzing, Static analysis, Dynamic analysis, Static and dynamic analysis, Hardware support and Tests. The analysis showed that most studies focus on static analysis and fuzzing tools. Categories such as Hardware support and Dynamic Analysis were less represented.

Figure 3, 4, 5, 6, 7 provides a detailed overview of publications in each category (fuzzing, static analysis, dynamic analysis, hardware support and tests) by year. These results show trends in research.

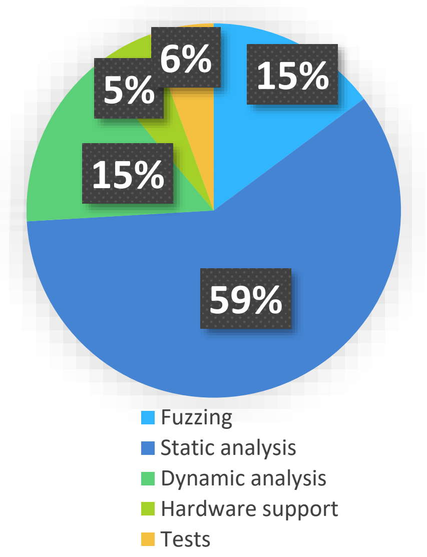

Figure 2: Overview of categories assigned to publications.

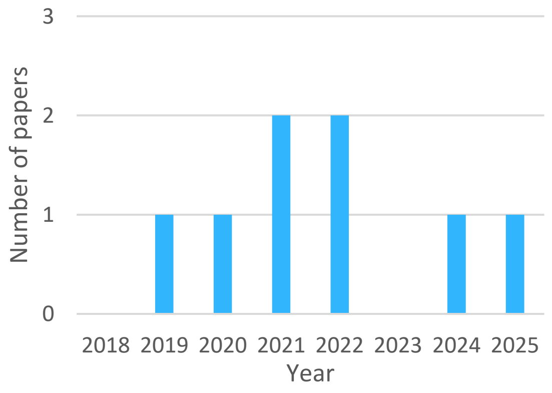

Figure 3: Number of papers for fuzzing.

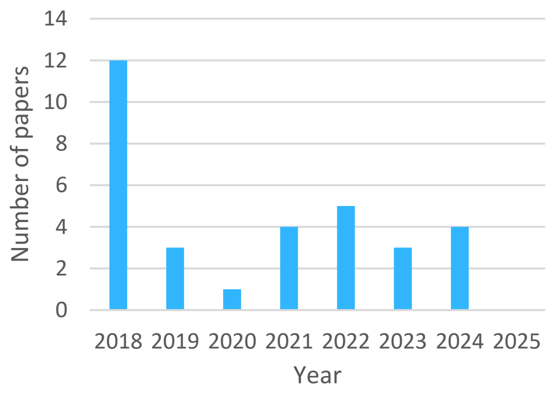

Figure 4: Number of papers for static analysis.

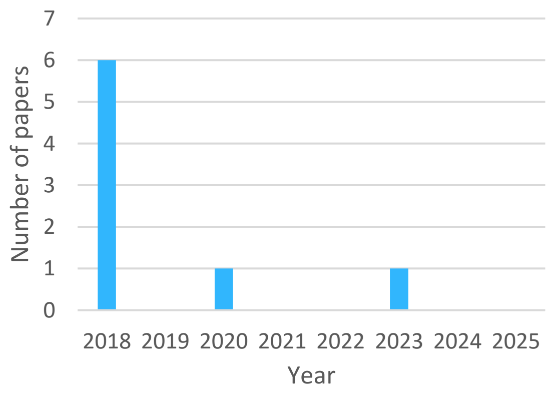

Figure 5: Number of papers for dynamic analysis.

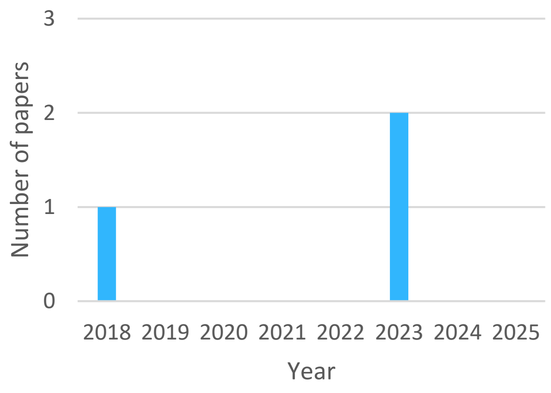

Figure 6: Number of papers for hardware support.

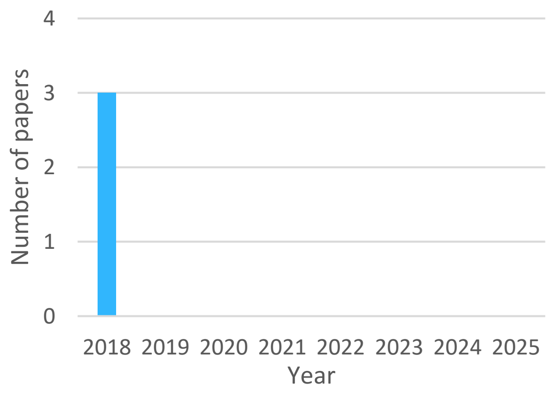

Figure 7: Number of papers for tests.

6. Conclusions

A systematic review of the literature has shown that static analysis is the most studied and used approach to detect security vulnerabilities in C++. This is due to its effectiveness in identifying vulnerabilities, such as buffer overflow or null pointer dereference, while still at the source code stage. As many as 32 publications were identified in this category, indicating its dominant role in ensuring software security. Dynamic analysis, fuzzing or hardware support, are less represented, which may suggest that their effectiveness is limited to specific scenarios or that they require further development and integration with modern tools. An analysis of the literature showed that publications related to the testing category were present only in the early years of the research period in question. This lack of new research in testing may indicate a declining interest in the subject or that testing as a method of ensuring security has been replaced by more advanced techniques.

The review data confirms that static analysis tools are now more popular and considered more effective in identifying security vulnerabilities in programs written in C++, which answers the research question. Static code analysis tools are more effective than others in detecting security vulnerabilities in programs written in C++.

However, dynamic analysis plays an important role in detecting bugs that only become apparent while the program is running. It is worth noting that methods combining static and dynamic analysis (6 publications) may offer a more comprehensive approach, indicating the need for further research into hybrid solutions.

Based on the literature review, it can be concluded that static analysis should be a cornerstone in the C++ software security process. Its ability to identify vulnerabilities at an early stage of software development makes it a key tool in the hands of developers. However, it is worth noting the potential of hybrid approaches that combine static and dynamic analysis. Such solutions can significantly increase the effectiveness of security vulnerability detection by combining the advantages of both techniques and eliminating their limitations. Fuzzing also plays an important role as a complementary method, especially in the context of unusual vulnerabilities that are difficult to predict with traditional code analysis techniques. The development of tools using fuzzing in combination with static and dynamic analysis can bring tangible benefits in ensuring comprehensive software security.

The results of the review also point to the need to revisit the study of testing as an integral part of the security assurance process. Although testing appears to have become a less popular area of research in recent years, its role in verifying the correctness of systems in conjunction with other analysis techniques remains important. It is worth considering their use in environments where there are limitations to implementing advanced analysis tools.

The small number of publications on hardware support suggests that this area may require more attention in future research. The use of dedicated hardware support, combined with modern development tools, may open new possibilities for improving the efficiency and effectiveness of security features in programs written in C++.

References

[1] A. Al-boghdady, K. Wassif, M. El-ramly, The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT's Low-End Devices, Sensors 21(7) (2021) 1-22, https://doi.org/10.3390/S21072329.

[2] A. Sheikh, Buffer Overflow, Certified Ethical Hacker (CEH) Preparation Guide, Apress, Berkeley, 2021, https://doi.org/10.1007/978-1-4842-7258-9_14.

[3] Unvalidated Redirects and Forwards Cheat Sheet, https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html, [30.12.2024].

[4] Bjarne Stroustrup's FAQ, https://www.stroustrup.com/bs_faq.html, [18.01.2025].

[5] C++26, https://en.cppreference.com/w/cpp/26, [10.01.2025].

[6] Standardization - Current Status, https://isocpp.org/std/status, [10.01.2025].

[7] What are the most secure Programming languages?, https://www.mend.io/most-secure-programming-languages/, [30.12.2024].

[8] M.A. Butt, Z. Ajmal, Z.I. Khan, M. Idrees, Y. Javed, An In-Depth Survey of Bypassing Buffer Overflow Mitigation Techniques, Applied Sciences 12(13) (2022) 1-31, https://doi.org/10.3390/APP12136702.

[9] Memory leak, https://owasp.org/www-community/vulnerabilities/Memory_leak, [30.12.2024].

[10] W. Li, D. Xu, W. Wu, X. Gong, X. Xiang, Y. Wang, F. Gu, Q. Zeng, Memory access integrity: detecting fine-grained memory access errors in binary code, Cybersecurity 2 (2019) 1–18, https://doi.org/10.1186/S42400-019-0035-X.

[11] Integer overflow, https://cplusplus.com/articles/DE18T05o/, [30.12.2024].

[12] M. Alqaradaghi, G. Morse, T. Kozsik, Detecting security vulnerabilities with static analysis – A case study, Pollack Periodica 17(2) (2021) 1–7, https://doi.org/10.1556/606.2021.00454.

[13] F. Pastore, L. Mariani, A. Goffi, M. Oriol, M. Wahler, Dynamic analysis of upgrades in C/C++ software, In IEEE 23rd International Symposium on Software Reliability Engineering (2012) 91–100, https://doi.org/10.1109/ISSRE.2012.9.

[14] Best Practices for Secure Programming in C++, https://www.mayhem.security/blog/best-practices-for-secure-programming-in-c, [30.12.2024].

[15] M.I. Mihailescu, S.L. Nita, Secure Coding Guidelines, Pro Cryptography and Cryptanalysis with C++23, Apress, Berkeley, 2023, https://doi.org/10.1007/978-1-4842-9450-5_7.

[16] X. Zhao, H. Qu, J. Xu, X. Li, W. Lv, G.G. Wang, A systematic review of fuzzing, Soft Computing 28 (2024) 5493–5522, https://doi.org/10.1007/S00500-023-09306-2.

[17] M. Skublewska-Paszkowska, M. Milosz, P. Powroznik, E. Lukasik, 3D technologies for intangible cultural heritage preservation—literature review for selected databases,

Heritage Science 10 (2022) 1–24, https://doi.org/10.1186/s40494-021-00633-x.

[18] D. Blackwell, I. Becker, D. Clark, Hyperfuzzing: black-box security hypertesting with a grey-box fuzzer, Empirical Software Engineering 30 (2025) 1-28, https://doi.org/10.1007/S10664-024-10556-3.

[19] K. Song, M.R. Gadelha, F. Brauße, R.S. Menezes, L.C. Cordeiro, ESBMC v7.3: Model Checking C++ Programs Using Clang AST, Lecture Notes in Computer Science 14414 (2024) 141–152, https://doi.org/10.1007/978-3-031-49342-3_9.

[20] C. Seas, G. Fitzpatrick, J.A. Hamilton, M.C. Carlisle, Automated Vulnerability Detection in Source Code Using Deep Representation Learning, In IEEE 14th Annual Computing and Communication Workshop and Conference (2024) 484– 490, https://doi.org/10.1109/CCWC60891.2024.10427574.

[21] K. Alshmrany, M. Aldughaim, A. Bhayat, L.C. Cordeiro, FuSeBMC v4: Improving Code Coverage with Smart Seeds via BMC, Fuzzing and Static Analysis, Formal Aspects of Computing 36(2) (2024) 1–25, https://doi.org/10.1145/3665337.

[22] F. Kasten, P. Zieris, J. Horsch, Integrating Static Analyses for High-Precision Control-Flow Integrity, In RAID '24: Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses (2024) 419–434, https://doi.org/10.1145/3678890.3678920.

[23] J. Kaivo, T. Devine, Defending the Heap: Diagnosing Undefined Behavior in Dynamic Memory with jkmalloc, In International Conference on Computational Science and Computational Intelligence (2023) 1572–1577, https://doi.org/10.1109/CSCI62032.2023.00259.

[24] P. Liu, Y. Lu, W. Yang, M. Pan, VALAR: Streamlining Alarm Ranking in Static Analysis with Value-Flow Assisted Active Learning, In 38th IEEE/ACM International Conference on Automated Software Engineering (2023) 1940–1951, https://doi.org/10.1109/ASE56229.2023.00098.

[25] S.H. Park, R. Pai, T. Melham, A Formal CHERI-C Semantics for Verification, Lecture Notes in Computer Science 13993 (2023) 549–568, https://doi.org/10.1007/978-3-031-30823-9_28.

[26] Q. Shen, H. Sun, G. Meng, K. Chen, Y. Zhang, Detecting API Missing-Check Bugs Through Complete Cross Checking of Erroneous Returns, Lecture Notes in Computer Science 13837 (2023) 391–407, https://doi.org/10.1007/978-3-031-26553-2_21.

[27] K. Hohentanner, P. Zieris, J. Horsch, CryptSan: Leveraging ARM Pointer Authentication for Memory Safety in C/C++, In SAC '23: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (2023) 1530–1539, https://doi.org/10.1145/3555776.3577635.

[28] K.M. Alshmrany, M. Aldughaim, A. Bhayat, L.C. Cordeiro, FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing, Lecture Notes in Computer Science 13241 (2022) 336–340, https://doi.org/10.1007/978-3-030-99429-7_19.

[29] S. Godboley, A. Dutta, R.K. Pisipati, D.P. Mohapatra, SSG-AFL: Vulnerability detection for Reactive Systems using Static Seed Generator based AFL, In IEEE 46th Annual Computers, Software, and Applications Conference (2022) 1728–1733, https://doi.org/10.1109/COMPSAC54236.2022.00275.

[30] F.R. Monteiro, M.R. Gadelha, L.C. Cordeiro, Model checking C++ programs, Software Testing, Verification and Reliability 32(1) (2022) 1-30, https://doi.org/10.1002/stvr.1793.

[31] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities, IEEE Transactions on Dependable and Secure Computing 19(4) (2022) 2244–2258, https://doi.org/10.1109/TDSC.2021.3051525.

[32] A. Machiry, J. Kastner, M. McCutchen, A. Eline, K. Headley, M. Hicks, C to checked C by 3c, Proceedings of the ACM on Programming Languages 6(OOPSLA1) (2022) 1-29, https://doi.org/10.1145/3527322.

[33] K.M. Alshmrany, R.S. Menezes, M.R. Gadelha, L.C. Cordeiro, FuSeBMC: A White-Box Fuzzer for Finding Security Vulnerabilities in C Programs (Competition Contribution), Lecture Notes in Computer Science 12649 (2021) 363–367, https://doi.org/10.1007/978-3-030-71500-7_19.

[34] S. Zaharia, T. Rebedea, Ș. Trăușăn-Matu, CWE Pattern Identification using Semantical Clustering of Programming Language Keywords, In 23rd International Conference on Control Systems and Computer Science (2021) 119–126, https://doi.org/10.1109/CSCS52396.2021.00027.

[35] K.M. Alshmrany, M. Aldughaim, A. Bhayat, L.C. Cordeiro, FuSeBMC: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs, Lecture Notes in Computer Science 12740 (2021) 85–105, https://doi.org/10.1007/978-3-030-79379-1_6.

[36] Y. Iqbal, M.A. Sindhu, M.H. Arif, M.A. Javed, Enhancement in Buffer Overflow (BOF) Detection Capability of Cppcheck Static Analysis Tool, In International Conference on Cyber Warfare and Security (2021) 112–117, https://doi.org/10.1109/ICCWS53234.2021.9703043.

[37] F.J. Gao, Y. Wang, L.Z. Wang, Z. Yang, X.D. Li, Automatic Buffer Overflow Warning Validation, Journal of Computer Science and Technology 35 (2020) 1406–1427, https://doi.org/10.1007/S11390-020-0525-Z.

[38] Y. Rong, P. Chen, H. Chen, Integrity: Finding integer errors by targeted fuzzing, Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering 335 (2020) 360–380, https://doi.org/10.1007/978-3-030-63086-7_20.

[39] J. Giet, L. Mauborgne, D. Kästner, C. Ferdinand, Towards zero alarms in sound static analysis of finite state machines, Lecture Notes in Computer Science 11698 (2019) 3–18, https://doi.org/10.1007/978-3-030-26601-1_1.

[40] C. Zou, Y. Sui, H. Yan, J. Xue, TCD: Statically Detecting Type Confusion Errors in C++ Programs, In IEEE 30th International Symposium on Software Reliability Engineering (2019) 292–302, https://doi.org/10.1109/ISSRE.2019.00037.

[41] X. Liu, X. Li, R. Prajapati, D. Wu, DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing, In 33rd AAAI Conference on Artificial Intelligence (2019) 1044-1051, https://doi.org/10.1609/aaai.v33i01.33011044.

[42] Z. Gu, J. Wu, J. Liu, M. Zhou, M. Gu, An Empirical Study on API-Misuse Bugs in Open-Source C Programs, In IEEE 43rd Annual Computer Software and Applications Conference (2019) 11–20, https://doi.org/10.1109/COMPSAC.2019.00012.

[43] B. Lu, W. Dong, L. Yin, L. Zhang, Evaluating and Integrating Diverse Bug Finders for Effective Program Analysis, Lecture Notes in Computer Science 11293 (2018) 51–67, https://doi.org/10.1007/978-3-030-04272-1_4.

[44] M. Hermeling, Software Input Validation Secure and Pro-active Protection Against Hacker Attacks, ATZelektronik Worldwide 13 (2018) 26–31, https://doi.org/10.1007/S38314-017-0096-0.

[45] X.B.D. Le, F. Thung, D. Lo, C. Le Goues, Overfitting in semantics-based automated program repair, Empirical Software Engineering 23 (2018) 3007–3033, https://doi.org/10.1007/S10664-017-9577-2.

[46] A.Y. Gerasimov, L.V. Kruglov, M.K. Ermakov, S.P. Vartanov, An Approach to Reachability Determination for Static Analysis Defects with the Help of Dynamic Symbolic Execution, Programming and Computer Software 44 (2018) 467–475, https://doi.org/10.1134/S0361768818060051.

[47] F. Biondi, M.A. Enescu, A. Heuser, A. Legay, K.S. Meel, J. Quilbeuf, Scalable approximation of quantitative information flow in programs, Lecture Notes in Computer Science 10747 (2018) 71–93, https://doi.org/10.1007/978-3-319-73721-8_4.

[48] M. Dimjašević, F. Howar, K. Luckow, Z. Rakamarić, Study of integrating random and symbolic testing for object-oriented software, Lecture Notes in Computer Science 11023 (2018) 89–109, https://doi.org/10.1007/978-3-319-98938-9_6.

[49] F. Maurica, D.R. Cok, J. Signoles, Runtime Assertion Checking and Static Verification: Collaborative Partners, Lecture Notes in Computer Science 11245 (2018) 75–91, https://doi.org/10.1007/978-3-030-03421-4_6.

[50] A.Y. Gerasimov, Directed Dynamic Symbolic Execution for Static Analysis Warnings Confirmation, Programming and Computer Software 44 (2018) 316–323, https://doi.org/10.1134/S036176881805002X.

[51] M. Ye, J. Sherman, W. Srisa-An, S. Wei, TZSlicer: Security-aware dynamic program slicing for hardware isolation, In IEEE International Symposium on Hardware Oriented Security and Trust (2018) 17–24, https://doi.org/10.1109/HST.2018.8383886.

[52] A. Bican, R. Deaconescu, W.N. Chin, Q.T. Ta, Verification of C Buffer Overflows in C Programs, In 17th RoEduNet Conference: Networking in Education and Research (2018) 1–6, https://doi.org/10.1109/ROEDUNET.2018.8514126.

[53] K. Vorobyov, N. Kosmatov, J. Signoles, Detection of security vulnerabilities in C code using runtime verification: An experience report, Lecture Notes in Computer Science 10889 (2018) 139–156, https://doi.org/10.1007/978-3-319-92994-1_8.

[54] B. Cook, K. Khazem, D. Kroening, S. Tasiran, M. Tautschnig, M.R. Tuttle, Model checking boot code from AWS data centers, Lecture Notes in Computer Science 10982 (2018) 467–486, https://doi.org/10.1007/978-3-319-96142-2_28.

[55] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, F. Xie, CRETE: A versatile binary-level concolic testing framework, Lecture Notes in Computer Science 10802 (2018) 281–298, https://doi.org/10.1007/978-3-319-89363-1_16.