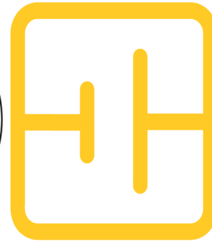

PPGEE
Programa de Pós-Graduação em Engenharia Elétrica - UFAM

FEDERAL UNIVERSITY OF AMAZONAS - UFAM

FACULTY OF TECHNOLOGY - FT

POSTGRADUATE PROGRAM IN ELECTRICAL ENGINEERING - PPGEE

LSVerifier: A BMC Approach to Identify Security Vulnerabilities in C Open-Source Software

Janislley Oliveira de Sousa

Manaus - AM

December 2023

Janislley Oliveira de Sousa

LSVerifier: A BMC Approach to Identify Security Vulnerabilities in C Open-Source Software

A dissertation submitted to the postgraduate program in Electrical Engineering, in the field of Modern Control and Automation Systems, at the Federal University of Amazonas, in fulfillment of the requirements for the Master of Science degree.

Supervisor:

Lucas Carvalho Cordeiro, Dr.

Co-Supervisor:

Eddie Batista de Lima Filho, Dr.

Federal University of Amazonas - UFAM

Faculty of Technology - FT

Manaus - AM

December 2023

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

S725l

Sousa, Janislley Oliveira de
LSVerifier: a BMC approach to identify security vulnerabilities in C open-source software / Janislley Oliveira de Sousa . 2023
94 f.: il.; 31 cm.

Orientadora: Lucas Carvalho Cordeiro
Coorientadora: Eddie Batista de Lima Filho
Dissertação (Mestrado em Engenharia Elétrica) - Universidade Federal do Amazonas.

1. Bounded model checking. 2. Software verification. 3. Security vulnerabilities. 4. Open-source software. 5. Large systems. I. Cordeiro, Lucas Carvalho. II. Universidade Federal do Amazonas III. Título

Poder Executivo
Ministério da Educação
Universidade Federal do Amazonas
Faculdade de Tecnologia
Programa de Pós graduação em Engenharia Elétrica

JANISLEY OLIVEIRA DE SOUSA

LSVERIFIER: A BMC APPROACH TO IDENTIFY SECURITY VULNERABILITIES IN C OPEN-SOURCE SOFTWARE PROJECTS

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Amazonas, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica na área de concentração Controle e Automação de Sistemas.

Aprovada em 20 de dezembro de 2023.

BANCA EXAMINADORA

Prof. Dr. Lucas Carvalho Cordeiro
Presidente
Universidade Federal do Amazonas

Prof. Dr. Leandro Buss Becker, Membro
Universidade Federal de Santa Catarina

Prof. Dr. Raimundo da Silva Barreto, Membro
Universidade Federal do Amazonas

Pós-Graduação em Engenharia Elétrica.
Av. General Rodrigo Octávio Jordão Ramos, nº 3.000 - Campus Universitário, Setor Norte - Coroado, Pavilhão do CETELI.
Fone/Fax (92) 99271-8954 Ramal:2607. E-mail: ppgee@ufam.edu.br

To all those who have supported me on my journey
as a scientist, this work is dedicated to you.

Acknowledgements

I am profoundly grateful for the guidance and support of my supervisor, Dr. Lucas Cordeiro, over the past three years. His unwavering belief in my abilities and his expertise has significantly shaped my professional and academic development. Being a part of Dr. Cordeiro's research group has been an immense privilege, and his mentorship has been instrumental in refining my research skills. His dedication and commitment to excellence have continually served as a source of inspiration.

Additionally, I would like to express my deep appreciation to my co-supervisor, Dr. Eddie Batista. His expertise and guidance have profoundly influenced my research. His knowledge in the field has greatly enhanced the quality of my work.

I extend my sincere thanks to Sidia Institute of Science and Technology for their support. The environment and resources they provided have been fundamental in the progression of my work. Additionally, I am deeply thankful to my colleagues at Sidia for their invaluable contributions. Their valuable insights have enriched my research, and their constant encouragement was a beacon of hope during challenging times.

My deepest gratitude goes to my family, whose love, support, and encouragement have been the foundation of my journey. I am particularly indebted to my parents for their role in my education and for always believing in me. This dissertation is dedicated to them, a token of my appreciation and love.

Finally, I want to express my heartfelt thanks to my beloved Vanessa Okawa, who has been a source of endless love and motivation. Her presence in my life has been a guiding light, helping me to persevere through every challenge. This work is also a dedication to her, symbolizing our shared journey and dreams.

Faça o teu melhor, na condição que você tem, enquanto você não tem condições melhores, para fazer melhor ainda.

Mário Sergio Cortella

LSVerifier: A BMC Approach to Identify Security Vulnerabilities in C Open-Source Software

Author:
Janislley Oliveira de Sousa
Supervisor:
Lucas Carvalho Cordeiro, Dr.
Co-Supervisor:
Eddie Batista de Lima Filho, Dr.

Abstract

This research advances the field of software vulnerability analysis by highlighting the critical role of software validation and verification techniques in developing systems with high dependability and reliability. A particular focus is placed on addressing the prevalent issue of memory safety properties in C software. We introduce LSVerifier, an innovative tool that utilizes the bounded model checking technique to uncover security vulnerabilities within C open-source software efficiently. LSVerifier stands out by not only identifying vulnerabilities but also producing a comprehensive report that outlines detected software weaknesses, thereby serving as a resource for developers aiming to enhance software security. Our experimental evaluation showcases the tool's effectiveness in scrutinizing large software systems while maintaining low peak memory usage. We applied LSVerifier to twelve open-source C projects, successfully detecting real software vulnerabilities that were later acknowledged and confirmed by the developers. The findings of this study underscore the potential of LSVerifier as a key instrument in the ongoing effort to secure open-source software against vulnerabilities.

Keywords: Bounded model checking, Software verification, Security vulnerabilities, Open-source software, Large systems.

List of Figures

Figura 1 – FuSeBMC: White-Box Fuzzing Framework for C Programs 24
Figura 2 – ESBMC architecture 26
Figura 3 – ESBMC's Performance in SV-COMP 2023 27
Figura 4 – This study's literature review examined research from the past decade on using model checking tools to verify vulnerabilities in open-source software. 34
Figura 5 – Structural complexities in Open-Source Software for security verification 45
Figura 6 – An overview of the proposed verification process. 46
Figura 7 – Example of counterexample created by LSVerifier. 56
Figura 8 – The LSVerifier's output during a verification procedure for PuTTY. . 56
Figura 9 – The verification report generated by LSVerifier for PuTTY. 57

List of Tables

Tabela 1 – Search queries used in different databases for the literature review. . 33
Tabela 2 – Main tools used to check software vulnerabilities using model checking techniques and their applicability on open-source software . . . 35
Tabela 3 – The LSVerifier's Configuration Parameters 53
Tabela 4 – Open-source software projects verified by LSVerifier. 60
Tabela 5 – Software vulnerabilities detected by LSVerifier, correlating the number of code-property violations with the amount of C code files, external libraries, lines of code, functions verified, maximum necessary memory, and verification time 63
Tabela 6 – Software property violations found, using the chosen dataset, with LSVerifier and split into the 11 categories recognized by ESBMC . . . 66
Tabela 7 – The vulnerabilities identified by LSVerifier along with the corresponding CWE numbers for each property violation detected 67
Tabela 8 – The reported issues involved code property violations and were submitted to the repositories of the respective software projects. These reports were made for the developers to confirm the associated software vulnerabilities 71

LIST OF ABBREVIATIONS AND ACRONYMS

ABV - Array Bounds Violated

AF - Assertion Failure

ALB - Array Lower Bound

AOOB - Access to Object Out of Bounds

AST - Abstract Syntax Tree

AUB - Array Upper Bound

CBMC - C Bounded Model Checker

CFG - Control-Flow Graph

CTL - Computation Tree Logic

CWE - Common Weakness Enumeration

DZ - Division by Zero

ESBMC - Efficient SMT-Based Context-Bounded Model Checker

GBF - Gray-Box Fuzzing

IDO - Invalidated Dynamic Object

IP - Invalid Pointer

IPF - Invalid Pointer Freed

IR - Intermediate Representation

MC - Bounded Model Checking

NP - NULL Pointer

NaN - Not-a-Number

SAT - Boolean Satisfiability

SMT - Satisfiability Modulo Theories

SOV - Same Object Violation

SSA - Static Single Assignment

SV-COMP - International Competition on Software Verification

List of publications

1. **[SBSeg 2023 - Qualis A4][Conference - Published]** de Sousa, J. O., de Farias, B. C., da Silva, T. A., de Lima Filho, E. B., & Cordeiro, L. C. LSVerifier: A BMC Approach to Identify Security Vulnerabilities in C Open-Source Software Projects. In XXIII Brazilian Symposium on Information and Computational Systems Security. DOI: <https://doi.org/10.5753/sbseg_estendido.2023.235802>

2. **[STTT - Qualis A4][Journal - Submitted]** de Sousa, J. O., de Farias, B. C., da Silva, T. A., & Cordeiro, L. C. (2023). Finding Software Vulnerabilities in Open-Source C Projects via Bounded Model Checking. ArXiv preprint: <https://arxiv.org/abs/2311.05281>

3. **[SBMF 2023 - AFRiTS][Workshop - Presented]** de Sousa, J. O. Memory Safety in Linux Kernel Drivers: Enhancing Security with Formal Verification. Workshop on Automated Formal Reasoning for Trustworthy AI Systems.

Awards

LSVerifier received the Best Tool Paper Award at SBSeg 2023.

Contents

1 INTRODUCTION 14
1.1 Motivation 15
1.2 Problem Definition 16
1.3 Objectives 17
1.4 Contributions 18
1.5 Outline 18

2 BACKGROUND THEORY 20
2.1 Vulnerabilities in Open-Source Software Projects 20
2.2 Formal Verification Techniques 22
2.2.1 Static Analysis 22
2.2.2 SMT Solvers 22
2.2.3 Bounded Model Checking 23
2.2.4 Fuzzing 24
2.2.5 Abstract Interpretation 24
2.3 ESBMC 25
2.4 Common Software Vulnerabilities in C Programming Language . . . 27
2.5 Summary 30

3 RELATED WORK 32
3.1 Review of Literature 32
3.2 Formal Verification Tools for Analyzing Open-Source Software Security Vulnerabilities 36
3.3 Summary 42

4 THE PROPOSED VERIFICATION METHODOLOGY 44

4.1 Architectural Patterns of Open-Source Software for Effective Vulnerability Verification 44
4.2 The Development of Proposed Methodology 46
4.3 Architecture and Main Functionalities 51
4.3.1 The LSVerifier's Tool Implementation 52
4.3.2 File Listing 53
4.3.3 Function Listing and Prioritization 53
4.3.4 Exporting results 54
4.4 An Illustrative Examples of Using LSVerifier 54
4.5 Summary 57

5 EXPERIMENTAL EVALUATION 59
5.1 Experimental Setup 59
5.2 Experimental Objectives 61
5.3 Threats to the Validity of Experiments 61
5.3.1 Benchmark selection 61
5.3.2 Performance and correctness 62
5.3.3 Counterexample validation 62
5.4 Experimental Results 62
5.5 Violated Properties Analysis 70
5.6 Summary 78

6 CONCLUSIONS 81
6.1 Future Works 83

Bibliography 84

1

INTRODUCTION

Developing software that is both secure and devoid of bugs presents a multifaceted and highly intricate challenge, especially in the context of an increasingly connected and digitized world (RODRIGUEZ; PIATTINI; EBERT, 2019; CORDEIRO; FILHO; BESSA, 2020). The implications of software vulnerabilities are not merely confined to technical malfunctions but extend to potentially catastrophic consequences (CORDEIRO; FILHO, 2016; TIHANYI et al., 2023). Airbus discovered a software vulnerability in the A400M aircraft, leading to a crash in 2015 (GADELHA; CORDEIRO; NICOLE, 2020). The fault in the engine control units caused the engines to power off shortly after take-off. Also, security researchers could remotely exploit a vulnerability in the Jeep Cherokee's Uconnect infotainment system (GREENBERG, 2015). By gaining access to the system, they could take over various vehicle functions, including engine and brakes. These examples highlight the growing importance of software integrity and security in embedded systems and the Internet of Things (IoT). Continuous monitoring, rigorous testing, adherence to best practices, and coordinated vulnerability disclosure are essential to mitigate these issues (MORSE et al., 2011; GADELHA; MENEZES; CORDEIRO, 2021).

For instance, in the C programming language (KERNIGHAN; RITCHIE, 2006), widely used to develop critical software, e.g., operating systems and drivers, execution of unsafe code might lead to undefined behaviors (VOROBYOV; KOSMATOV; SIGNOLES, 2018). This is a common cause of memory problems, including buffer overflows and double-free violations (CORDEIRO; FISCHER; MARQUES-SILVA, 2011; ALSHM-

RANY et al., 2022). Furthermore, these errors are some of the main threats to software security (VEEN et al., 2012) since attackers can exploit them to execute malicious code. Such an aspect is even worse in open-source software as the same attackers can quickly read code and easily find vulnerable spots (TAN et al., 2014). In addition, as this kind of project tends to be widely used by the general public, one can even question its very nature. Therefore, developers must use the available resources to validate source code more often. Finally, new advanced tools should be further developed to improve the associated software security (HOEPMAN; JACOBS, 2007; OHM et al., 2020). For instance, to demonstrate the importance of software validation and verification, one could mention the case that involved Log4j (DUCKLIN, 2021). In short, data was printed out or logged into a file that might be used to take over a server. It could be done because Log4j permitted the injection of external logging text, whose format and content could be chosen through look-ups. This way, sensitive data could be leaked, or a network connection could be made to acquire and run malicious code.

Numerous methods have been introduced to detect vulnerabilities in C programs (ALSHMRANY et al., 2021; GADELHA et al., 2019). For example, fuzzing techniques, such as black-, grey-, or white-box approaches, exploit random program inputs to identify unexpected behaviors (BÖHME et al., 2017; GODEFROID, 2020). Static analysis tools, like CPPCheck and Flawfinder, evaluate C programs for safety property violations (CADAR; DUNBAR; ENGLER, 2008; CORDEIRO; FISCHER, 2011; CLARKE; KROENING; LERDA, 2004a; GADELHA et al., 2019; PEREIRA; VIEIRA, 2020). Google's Address Sanitizer is another notable tool for identifying C program issues. Some strategies even combine static and dynamic verification (CORDEIRO et al., 2009). However, many of these techniques face challenges when applied to the large-scale software systems typical of open-source projects.

1.1 Motivation

Vulnerabilities pose significant threats to software security, providing attackers with opportunities to execute malicious code. It is crucial to assist software developers in

identifying subtle bugs within their code, such as array bounds violations, null-pointer dereferences, arithmetic overflows, and others (MEMARIAN et al., 2019). The open-source nature of many large-scale software systems amplifies this risk, as potential attackers can easily access and scrutinize the code to find vulnerabilities. The general public's widespread use of these projects highlights the need for developers to validate source code more frequently and thoroughly. Although modern methodologies like static and dynamic code analysis mitigate some risks, the unique challenges posed by the C language and the extensive use of third-party libraries require continuous vigilance. This context emphasizes the need for a collaborative approach within the open-source community to promote a culture of security best practices and continuous improvement. Consequently, developing and implementing advanced tools are essential for enhancing software security.

1.2 Problem Definition

The Common Weakness Enumeration (CWE) (MITRE, 2023) community often identifies vulnerabilities in programming languages, including C, and third-party libraries used across various open-source projects. The C language is known for being powerful but also tricky to use safely, especially regarding memory management. The C programming language's low-level nature and absence of safety checks make it susceptible to vulnerabilities such as buffer overflows, memory leaks, and insecure library usage (OORSCHOT, 2023). Open-source projects, which may lack regular maintenance or expert review, are particularly at risk. Modern tools and practices like static and dynamic code analysis can mitigate some risks. Still, the complexity of C combined with the broad use of third-party libraries claims for continuous vigilance in identifying and rectifying vulnerabilities. The collaborative effort within the open-source community is crucial to address these issues, highlighting the ongoing need for attention to security best practices.

One may also notice that large software systems are frequently composed of many elements declared in several source files, usually split across different directories.

This presents a challenge when applying static analysis tools to such systems: software model checkers typically verify a single file using a predetermined entry point (BEYER, 2022). Therefore, to manage vast software with multiple files, a common scenario in open-source applications, each file must be verified individually, adjusting the entry point as needed. Moreover, elements with varying priorities might need to be addressed appropriately.

1.3 Objectives

This work aims to demonstrate the efficiency and effectiveness of formal verification techniques to exploit security vulnerability issues in C open-source software projects, using Bounded Model Checking (BMC) and Satisfiability Modulo Theories (SMT) to identify security issues. We propose a pragmatic approach to verify large software systems: state-of-the-art bounded model checkers and parameters to be configured by users according to the vulnerability classes they want to check. We systematically guide an underlying verifier through source-code files to recursively explore threats in entire source-code directories or specific locations, e.g., functions, according to a pre-defined priority. This general goal is correlated with the following specific ones:

1. Develop a comprehensive review of existing techniques for identifying software vulnerabilities in C open-source software, highlighting current gaps and laying the groundwork for our research;
2. Develop a robust methodology that combines the strengths of BMC and SMT techniques, specifically aimed at detecting and classifying security vulnerabilities in C open-source software;
3. Develop a novel verification tool that integrates input-code analysis with BMC techniques, tailored for vulnerability detection in large open-source software systems;
4. Design a comprehensive evaluation analysis using counterexamples to pinpoint findings and provide accessible reports for facilitating issue reproduction and

correction.

1.4 Contributions

The original contribution of this work is the development of an innovative methodology that integrates input-code analysis with Bounded Model Checking for identifying and evaluating software vulnerabilities in large-scale systems, particularly relevant to open-source C software projects. This methodology has been practically implemented and tested. The research further contributes:

1. A comprehensive review of techniques for detecting software vulnerabilities in open-source applications, identifying gaps, and setting the stage for future research;
2. The development of LSVerifier, a robust verification tool that integrates input-code analysis, prioritized function analysis, and BMC techniques for detecting vulnerabilities in extensive software systems;
3. An evaluation structure that assesses findings in a detailed and user-friendly report, which can be used for problem reproduction and correction;
4. A thorough evaluation of the methodology across a wide range of open-source applications, demonstrating its efficacy in finding real vulnerabilities like overflows, array-out-of-bounds, divisions by zero, and pointer-safety issues.

1.5 Outline

The structure of this document unfolds as follows. Chapter 1 briefly introduces BMC and the ESBMC architecture and describes the essential background theories of the SMT solvers and software vulnerabilities. Chapter 2 delves into the foundational concepts pivotal for the LSVerifier, particularly emphasizing state-of-the-art BMC and SMT verification tools tailored for C open-source projects. In Chapter 3, we traverse the literature

on C-based software verification and the significance of BMC tools in this realm. Chapter 4 is segmented into several sections, detailing our core contributions: the introduction of our verification methodology for large software systems, the architecture of LSVerifier, and the software organization that inspired its inception. Chapter 5 showcases our experimental findings, encompassing the setup, objectives, and a comprehensive discussion on the results. Chapter 6 contains the conclusion and future directions.

2

Background Theory

In this section, we delve into the foundational concepts and technologies essential for the LSVerifier tool. Specifically, we focus on establishing a systematic framework for state-of-the-art BMC and SMT verification tools, highlighting their relevance and optimization for C open-source software projects. Furthermore, we outline the methodology formulated to leverage these techniques in detecting and classifying security vulnerabilities, ensuring its alignment with the unique characteristics and challenges posed by C open-source projects.

2.1 Vulnerabilities in Open-Source Software Projects

Open-source software is characterized by its publicly accessible source code, allowing anyone to inspect, modify, and distribute it. This model fosters a collaborative environment where developers worldwide contribute to software projects. Despite its benefits in innovation and community development, open-source software often faces increased security risks due to its openness and varied maintenance practices. Developing effective vulnerability analysis tools has become crucial as such software's vulnerabilities have surged (WEN, 2017; PLATE; PONTA; SABETTA, 2015; MUEGGE; MURSHED, 2018; ALHAWI; MUSTAFA; CORDEIRO, 2019; MATULEVICIUS; CORDEIRO, 2021).

Open-source projects are particularly vulnerable due to their collaborative nature, underscoring the need for effective security tools. Research indicates these tools can help developers create more secure systems. However, challenges remain, such

as ensuring software security throughout the development process and addressing vulnerabilities that arise from complex dependencies and supply chain issues, including manufacturer-reserved backdoors and third-party libraries. Xiao et al. (XIAO; WITSCHEY; MURPHY-HILL, 2014) highlight that security vulnerabilities pose significant challenges in open-source software development, influenced by various social factors. Their research suggests that security tools can aid developers in creating safer software by identifying and addressing vulnerabilities during the development phase. However, issues like a lack of regular maintenance and understanding of the consequences of security failures contribute to these vulnerabilities. Zou et al. (ZOU et al., 2019) emphasize the importance of checking open-source software for supply chain issues and hidden vulnerabilities in third-party components. They note that despite developers' awareness of secure coding practices, ensuring comprehensive security remains challenging. Moreover, some problems may still exist in the available code as it is challenging to detect security risks before software deployment (GUEYE et al., 2021).

New tools and methods like static and dynamic analyzers have been developed to address these challenges (PONTA; PLATE; SABETTA, 2020), underscoring the critical need for improved security in open-source software development. Palmskog et al. (PALMSKOG; CELIK; GLIGORIC, 2018) introduced piCoq, a tool for detecting vulnerabilities in large-scale projects, focusing on dependency tracking and parallel checking. Similarly, Ruscio et al. (RUSCIO; PELLICCIONE; PIERANTONIO, 2012) developed EVOSS, a tool for identifying system configuration inconsistencies, proven effective in Linux distributions like Debian and Ubuntu. These advancements underscore the importance of security in open-source development, highlighting the risks associated with open code access and the potential impacts of vulnerabilities due to code reuse and dependencies in the software industry (PLATE; PONTA; SABETTA, 2015).

The open nature of these projects, beneficial for rapid development and wide access, also significantly increases the risk of intentional and unintentional vulnerabilities in critical and large-scale open-source software. This scenario highlights the urgent need for new tools and methods to address and mitigate these emerging security challenges.

2.2 Formal Verification Techniques

Manual inspection of complex software is often error-prone and costly, necessitating effective tool support. While several tools use test vectors to examine specific software executions and uncover design flaws, formal verification tools offer a more comprehensive approach. They can check a design's behavior against all possible inputs, ensuring a thorough analysis. Additionally, techniques like model checking, fuzzing, abstract interpretation, and interpolation provide alternative strategies for identifying software vulnerabilities (CORDEIRO; FILHO; BESSA, 2020), each contributing uniquely to the reliability and security of software systems.

2.2.1 Static Analysis

Static analysis involves assessing a codebase for potential errors, vulnerabilities, or deviations and effectively identifying common coding problems before a program's release (GADELHA et al., 2019). These tools analyze a program's code statically, without execution, and work on source code and compiled forms, though decoding the latter can be challenging. However, early static analysis methods were not recursive, meaning they couldn't fully understand the implications of self-calling functions or procedures (GOSEVA-POPSTOJANOVA; PERHINSCHI, 2015). While static analysis detects fine-grained bugs and adheres to coding standards, it cannot address every security issue or offer design advice. Its limitations became apparent as software complexity increased, particularly with multi-procedure structures and recursive patterns.

2.2.2 SMT Solvers

Solvers are designed to determine the satisfiability of formulas, broadly categorized into Boolean satisfiability (SAT) and satisfiability modulo theories (SMT). SAT solvers, utilizing methods like the Davis-Putnam-Logemann-Loveland (DPLL) and the more efficient Conflict Driven Clause Learning (CDCL), focus on validating Boolean formulas. However, translating higher-level system designs into Boolean logic can be resource-

intensive, leading to the development of SMT solvers. These solvers handle more abstract levels and apply rules from mathematical theories like Equality, Bit-vector, Linear Arithmetic, and Arrays, representing the field's evolution.

In this work, various modern SMT solvers are utilized. Yices (DUTERTRE; MOURA, 2006), developed by SRI International, handles a range of first-order theories useful for both software and hardware representations. It efficiently manages complex formulas across various theories defined in SMT-LIB. CVC4 (BARRETT et al., 2011), a collaboration between NYU and the University of Iowa, integrates features of CVC3 and SMT-LIBv2, benefiting from system architecture and decision procedure advancements. Z3 (MOURA; BJØRNER, 2008), created by Microsoft Research, is tailored for software verification and analysis, and used in several tools. Lastly, Boolector (BRUMMAYER; BIERE, 2009) specializes in the quantifier-free theory of bit-vectors and arrays, employing term rewriting and bit-blasting techniques.

2.2.3 Bounded Model Checking

BMC is a verification technique that detects errors up to a specified depth k by employing boolean satisfiability (SAT) or SMT. However, without a known upper bound for k, BMC cannot guarantee complete system correctness. It only explores a limited state space by unwinding loops and recursive functions to a maximum depth. This bounded nature of BMC makes it effective for uncovering fundamental errors in applications (CLARKE; KROENING; LERDA, 2004b; MERZ; FALKE; SINZ, 2012a; GADELHA et al., 2019; IVANCIC et al., 2005), and properties under verification are defined as follows:

$$\mathrm{BMC}_{\Phi}(k) = I(s_1) \wedge \left(\bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left(\bigvee_{i=1}^{k} \neg\phi(s_i) \right), \tag{2.1}$$

where, $I(s_1)$ is the set of initial states for a system; $\bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$ is the transition relation between time steps i and $i+1$, encompassing the evolution of the system over k steps; and $\bigvee_{i=1}^{k} \neg\phi(s_i)$ represents the negation of the property ϕ at state s_i, indicating a violation of the given property within a bound k. Together, these components formulate

a problem that is satisfiable if and only if a counterexample of length k or less exists, implying a violation of the specified property within the given bound.

2.2.4 Fuzzing

Fuzzing is a software testing technique utilized to discover vulnerabilities in software systems. It involves generating random or semi-random inputs to a C program, exploiting the fact that many critical security flaws arise from inadequate input validation (ALSHMRANY et al., 2021). The random nature of fuzzing inputs makes it likely they'll be unexpected or improper in a target program. If the program fails to reject these inputs, it may hang or crash, indicating potential security weaknesses. Fuzzing is an efficient, cost-effective method for identifying security vulnerabilities in C programs, with software unable to withstand fuzz testing being particularly prone to security issues. Figure 1 shows the FuSeBMC tool combining fuzzing with the BMC technique.

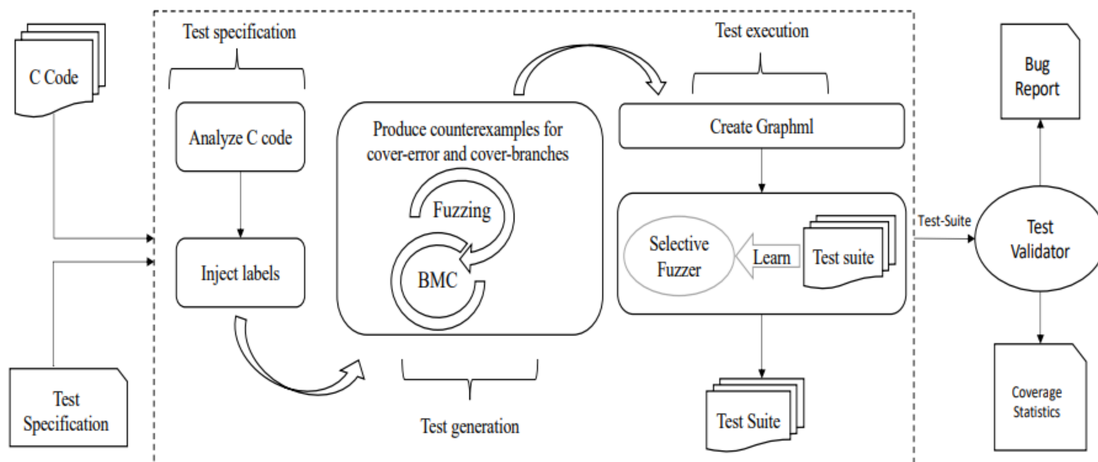

Figure 1 – FuSeBMC: White-Box Fuzzing Framework for C Programs

2.2.5 Abstract Interpretation

Abstract interpretation is a theory for abstracting and approximating mathematical structures in programming languages to infer or verify undecidable program properties (COUSOT, 2012). It begins with formally defining a language's semantics, formalizing

program properties, and expressing the key property in fixed point form. The theory provides methods for abstracting properties and fixed points, leading to verified abstract semantics where only relevant properties are preserved. Verification is done by checking fixed points inductively, and for property inference in static analyzers, fixed point approximation with widening/narrowing is used. Since verification for infinite systems is undecidable, abstractions may be over-approximating, leading to potential false alarms, which require refinement to distinguish between real and fake executions. The precision and scalability of abstractions are balanced by refining or coarsening them, with abstract interpreters and domains adjusting the precision/cost ratio (MENEZES et al., 2023).

2.3 ESBMC

As employed in this study, the Efficient SMT-based Context-Bounded Model Checker (ESBMC) (GADELHA; MENEZES; CORDEIRO, 2021) is a robust and openly available tool that serves as our chosen BMC module for software verification. This mature model checker is designed to verify programs written in C/C++, Kotlin, and Solidity. ESBMC is equipped to automatically assess pre-defined safety properties and user-specified assertions within programs, whose safety properties cover a range of concerns such as array out-of-bounds, illegal pointer dereferences, integer overflows, and division by zero. Additionally, ESBMC supports various language frontends, including Clang for C/C++ and Soot via Jimple for Java/Kotlin, and implements Solidity's grammar production rules for Ethereum's Solidity language. ESBMC is underpinned by state-of-the-art incremental BMC techniques and k-induction proof-rule algorithms rooted in SMT and constraint programming (CP) solvers. The ESBMC's prowess has been demonstrated in a variety of contexts. Indeed, it is recognized for its successful application in verifying single and multi-threaded code, effectively identifying intricate bugs in real-world software (CORDEIRO; FILHO, 2016).

Figure 2 shows the ESBMC architecture. White rectangles represent input and output and gray rectangles represent the verification steps. ESBMC uses several key

components during the verification process:

1. **Control-flow Graph (CFG) Generator:** For C++ programs, this component includes type-checking and static analysis covering various checks. It creates an Intermediate Representation (IR) for GOTO program generation. For ANSI-C, it converts AST into a GOTO program, adding various checks and simplifications.

2. **Symbolic Execution Engine:** This engine symbolically executes the GOTO program, unrolling loops, generating Static Single Assignments (SSA) forms, and deriving safety properties for SMT solvers. It includes pointer safety checks and simplifies the program using various techniques.

3. **SMT Back-end:** Supports multiple solvers and is adaptable for encoding quantifier-free formulas. It encodes the SSA form into a formula to check satisfiability and generates counterexamples if a bug is detected.

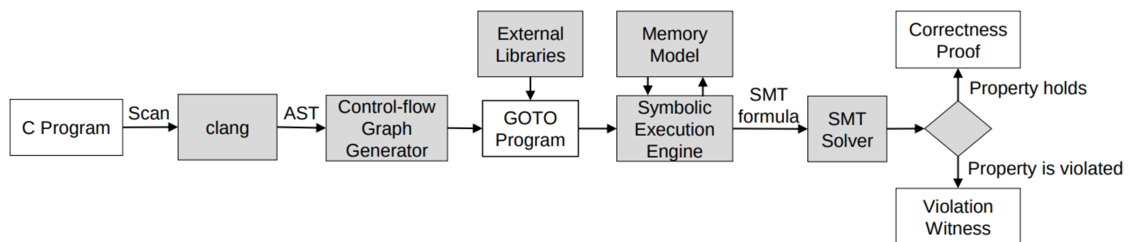

Figure 2 – ESBMC architecture

The fundamental methodology of BMC, as utilized by ESBMC, consists of unrolling a target system for a predetermined number of iterations and establishing a verification condition (VC). If this VC is satisfiable, it signifies the existence of a counterexample for a particular property at a specific depth. While ESBMC has proven effective in verifying software properties efficiently, the challenge of scaling BMC tools for extensive software evaluation persists due to resource limitations.

In the realm of software verification, ESBMC has been distinguished as one of the foremost BMC tools, as corroborated by its commendable performance in the recent editions of the International Competition on Software Verification (SV-COMP) (GADELHA;

MENEZES; CORDEIRO, 2021). Notably, ESBMC achieved fifth place among 44 state-of-the-art C software verifiers in the comprehensive ranking of SV-COMP 2023 (BEYER, 2023). Figure 3 illustrates ESBMC's performance in the competition.

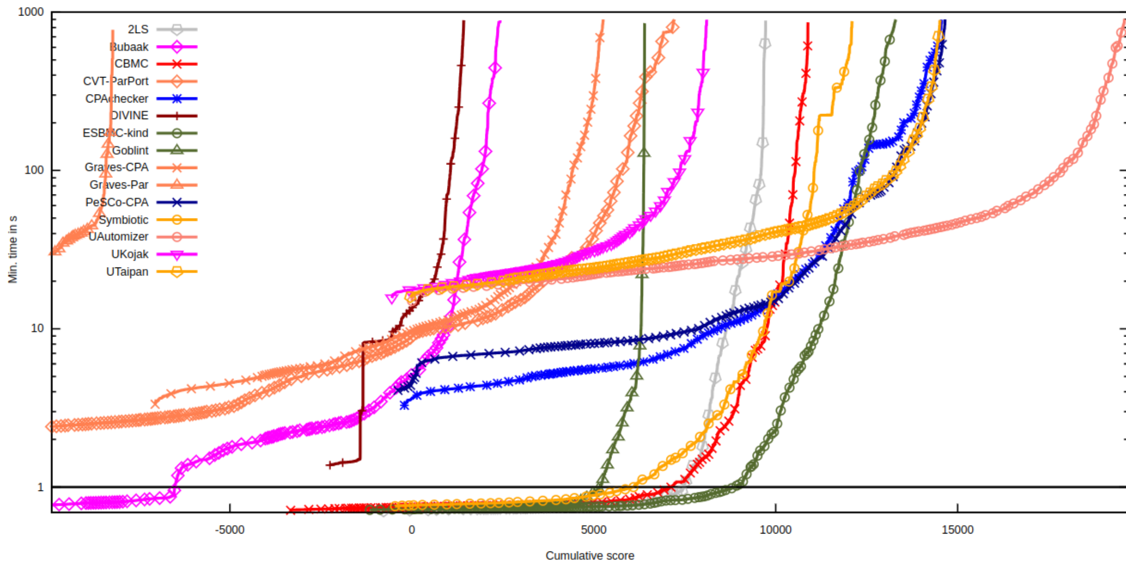

Figure 3 – ESBMC's Performance in SV-COMP 2023

2.4 Common Software Vulnerabilities in C Programming Language

As a formal definition, a software vulnerability is a security flaw, glitch, or weakness found in code that could be exploited by an attacker, leading, for instance, to sensitive-data leak or execution of malicious instructions (DEMPSEY et al., 2020). In addition, the common weakness enumeration (CWE) community (CORPORATION, 2019) identifies the most common vulnerabilities associated with the C/C++ programming language (CORPORATION, 2023). Here, we describe ten vulnerability categories that we consider in this work.

- **Definition 1 - Buffer Overflow:** This vulnerability is defined when copying data from one buffer to another without checking whether the former fits within the latter, independently of where they are located (i.e., heap, stack, etc.). Consequently, data in adjacent memory addresses get corrupted, which attackers can use to pro-

mote crash events, incorrect program behavior, information leakage, or execution of malicious code (CORPORATION, 2023). It is categorized under CWE-120 and defined as a buffer copy without checking the input size.

- **Definition 2 - Arithmetic Overflow:** This vulnerability is defined by an arithmetic operation's result surpassing the maximum capacity of its assigned data type. It is triggered when computations yield an integer overflow or wraparound, contradicting the assumption that the resultant value will invariably exceed the original. Such miscalculations can expose additional weaknesses, especially when used for resource allocation or execution control (CORPORATION, 2023). It is categorized under CWE-190 and defined as integer overflow or wraparound.

- **Definition 3 - Invalid Pointer:** This vulnerability category includes both dereferencing uninitialized (or null pointers) and deallocating memory using uninitialized or invalid pointers. In the case of dereferencing, the application accesses memory that it is not supposed to, often resulting in crashes or unexpected program behavior. This is closely related to CWE-476. In the case of deallocating memory, known as Invalid Pointer Freed, a program attempts to free a memory location using an uninitialized or invalid pointer, leading to potential corruption of memory and program instability. This specific issue is often related to CWE-416, which focuses on use-after-free scenarios. Both these forms of invalid pointer usage can lead to unpredictable program behavior and may be exploited by attackers (CORPORATION, 2023).

- **Definition 4 - Improper Buffer Access:** This vulnerability is defined when software uses a sequential operation to read or write a buffer with an incorrect length value. Consequently, memory outside of a buffer's bounds is accessed. This way, an attacker can access sensitive data or execute arbitrary code (CORPORATION, 2023). This software vulnerability falls under CWE-119 and is defined as an improper restriction of operations within the bounds of a memory buffer.

- **Definition 5 - Null Pointer Dereference:** This vulnerability is defined when an application dereferences a null pointer, often due to race conditions or program-

ming errors. Usually, it causes a crash or exit. An attack using it can aim at service denial (CORPORATION, 2023). It is categorized under CWE-476.

- **Definition 6 - Double Free:** This vulnerability is defined when a program calls *free()* twice with the same argument, which typically causes the corruption of memory management data structures. It happens because the first attempt allows the related memory space to be used by another part of the same program, and the second attempt releases something thought to be still in use. An attacker could access this buffer and execute arbitrary code or cause a crash (CORPORATION, 2023). It is categorized under CWE-415.

- **Definition 7 - Division by zero:** This vulnerability is defined when an unexpected value is returned to running code, even due to an undetected error, and then used in operations. If not handled, it often leads to a hardware trap, then finishing a given program. Moreover, as handling code is not deeply tested, in general, such a condition can be used by attackers (CORPORATION, 2023). It is categorized under CWE-369.

- **Definition 8 - Array bounds violated:** This vulnerability occurs when a program attempts to access an array element at an invalid index, either below zero or beyond an array's length, leading to data corruption, crashes, or unauthorized code execution (CORPORATION, 2023). It is categorized under CWE-787 and defined as modifying an index or performing pointer arithmetic that accesses a memory location outside a buffer's boundaries.

- **Definition 9 - Pointer arithmetic violation:** This vulnerability occurs when a product employs pointer arithmetic (e.g., subtraction, comparison) to ascertain size. It can lead to the same object violation when pointers from different memory blocks are used. In C programming, pointer arithmetic is commonly used to navigate arrays or compare memory positions, assuming the pointers involved reference the same allocated memory block. Issues arise when arithmetic operation is attempted between pointers not pointing to the same memory block, leading to

undefined and unreliable results. It is categorized under CWE-469 and defined as using pointer subtraction to determine size (CORPORATION, 2023).

- **Definition 10 - Assertion violation:** This vulnerability occurs when a condition provided to the function *assert* is not satisfied during program execution. A reachable assertion failure suggests the existence of a program execution path that leads to the assertion's location, where the variables' value does not meet the expected conditions. Assertions are often used to verify that program variables remain within user-defined bounds. An assertion failure may reveal logical errors that could be exploited, resulting in unpredictable behavior or system crashes. This type of vulnerability is associated with CWE-617, where assertions are expected to hold during normal execution (CORPORATION, 2023).

2.5 Summary

In this chapter, we have delved into various aspects and challenges of open-source software development, focusing particularly on the security vulnerabilities inherent in these projects. The collaborative and open nature of open-source software, while fostering innovation and community development, also exposes it to increased security risks. This situation underscores the necessity for effective and sophisticated security tools to navigate the complexities of open-source software, including issues like code dependencies and supply chain vulnerabilities.

We have explored various formal verification techniques, such as static analysis, SMT solvers, bounded model checking, fuzzing, and abstract interpretation. Each of these methodologies offers unique advantages in identifying and addressing vulnerabilities within software systems. Particularly, we discussed the ESBMC tool, a state-of-the-art model checker for C/C++, highlighting its capabilities to verify predefined safety properties and user-specified assertions in programs automatically. ESBMC's performance in the International Competition on Software Verification (SV-COMP) illustrates its effectiveness and reliability in software verification. It was chosen as the model checker for this work.

Finally, the chapter outlined ten major categories of software vulnerabilities, providing a comprehensive view of the security flaws in C/C++ programming environments. These include buffer overflow, arithmetic overflow, invalid pointer, improper buffer access, null pointer dereference, double free, division by zero, array bounds violation, pointer arithmetic violation, and assertion violation. Recognizing and understanding these vulnerabilities is crucial in developing strategies to mitigate them effectively. Our proposed methodology will cover all these software vulnerabilities.

In conclusion, studying open-source software vulnerabilities and developing verification tools are crucial in addressing the security challenges in open-source projects. The ongoing evolution of verification techniques and tools reflects the growing complexity of software systems and the ever-present need for more robust and efficient methods to ensure software security and reliability. This chapter sets the stage for the following sections, where we will apply these concepts and tools to analyze and secure C open-source software projects, aiming to bridge the gap in current security practices in open-source development.

3

Related Work

This section reviews the literature on verifying security vulnerabilities in C-based open-source software and discusses prevalent BMC tools used in verification processes. It also outlines the methodology for selecting these studies.

3.1 Review of Literature

The security of C-based open-source software is vulnerable to large-scale attacks that exploit weaknesses in millions of end-user systems (GADELHA et al., 2018). We reviewed pivotal and recent studies comprehensively to comprehend the prevailing strategies addressing these vulnerabilities. These studies adhere to open-source policies and BMC standards, encompassing public domain and industrial approaches to verifying safety properties in expansive open-source software. Our literature review was methodically curated, focusing on articles from prominent publishers and databases such as the Institute of Electrical and Electronics Engineers (IEEE), the Association for Computing Machinery (ACM), Science Direct, Springer, Scopus, Web of Science, and Google Scholar. Our selection was based on specific research criteria, emphasizing papers published between 2013 and 2023, which include journals, conference proceedings, and periodicals. The search's keystring is described in Table 1.

The methodology for our literature review is depicted in Figure 4. From an initial database search, 3551 studies were identified. After removing duplicates, 3107 studies were selected for further review. A significant portion, 3043 studies, were excluded

after a review of titles and abstracts revealed they were irrelevant to the research in formal verification, did not meet the search criteria, or the full text was unavailable. This process resulted in 64 papers being assessed for eligibility. Subsequently, exclusion criteria were applied to papers not focused on the bounded model checking approach or not related to the open-source domain. Ultimately, 30 papers were classified as relevant, with 27 focusing on BMC tools and three on research in open-source software.

Database	**Search Query**
IEEE	("All Metadata":Open Source) AND ("All Metadata":Security OR "All Metadata":Verification OR "All Metadata":Vulnerabilities OR "All Metadata":Violation OR "All Metadata":Fault) AND ("All Metadata":Software OR "All Metadata":Program OR "All Metadata":Application OR "All Metadata":Tool) AND ("All Metadata":model checking)
ACM	[Abstract: open source] AND [Abstract: model checking] AND [[Abstract: security] OR [Abstract: verification*] OR [Abstract: vulnerabilit*] OR [Abstract: violation] OR [Abstract: fault*]] AND [[Abstract: software] OR [Abstract: program] OR [Abstract: application] OR [Abstract: tool]] AND [Publication Date: (01/01/2013 TO 30/06/2023)]
Science Direct	Year: 2013-2023 Title, abstract, keywords: (model checking) AND (Security OR Verification OR Vulnerabilities OR Violation OR Fault) AND (Software OR Program OR Tool)
Web of Science	ALL=(model checking) AND ALL=(open source) AND ALL=(Security OR Verification OR Vulnerabilities OR Violation OR Fault) AND ALL=(Software OR Program OR Tool)
Scopus	ABS((model checking) AND (open source) AND (Security OR Verification OR Vulnerabilities OR Violation OR Fault) AND (Software OR Program OR Tool)) AND (LIMIT-TO (PUBYEAR,2023) OR LIMIT-TO (PUBYEAR,2022) OR LIMIT-TO (PUBYEAR,2021) OR LIMIT-TO (PUBYEAR,2020) OR LIMIT-TO (PUBYEAR,2019) OR LIMIT-TO (PUBYEAR,2018) OR LIMIT-TO (PUBYEAR,2017) OR LIMIT-TO (PUBYEAR,2016) OR LIMIT-TO (PUBYEAR,2015) OR LIMIT-TO (PUBYEAR,2014) OR LIMIT-TO (PUBYEAR,2013))

Table 1 – Search queries used in different databases for the literature review.

This review aims to deepen our understanding of bounded model checking techniques used in open-source software verification. We focus on tools that support vulnerability exploitation, software security assessment challenges, and future research directions. Our systematic approach details the stages of data collection from various studies on bounded model checking and open-source software. Overall, this review

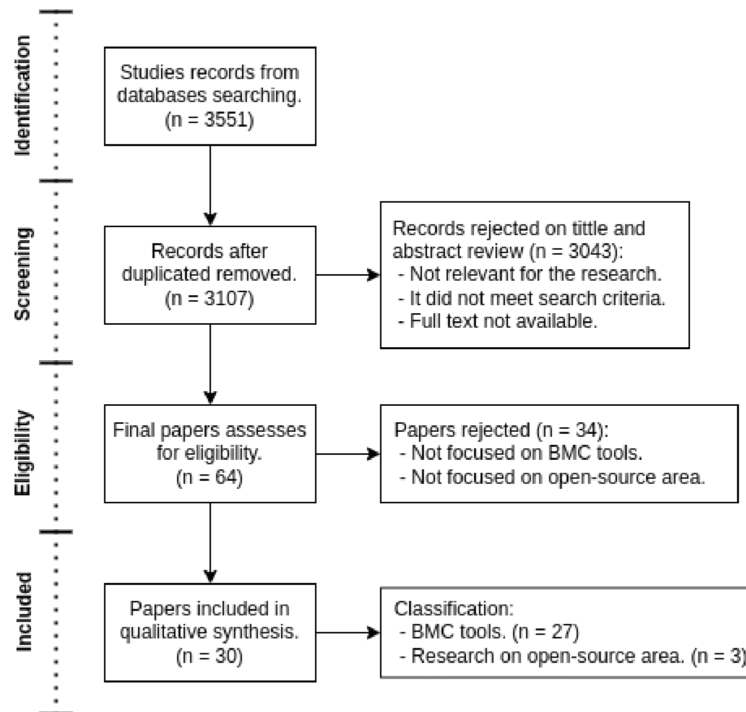

Figure 4 – This study's literature review examined research from the past decade on using model checking tools to verify vulnerabilities in open-source software.

highlights available bounded model checking tools and current research in the open-source domain. From our systematic search, we identified 30 papers for evaluation.

By evaluating identified papers and studies, we quantified the effectiveness of various security methods and pinpointed weaknesses and flaws that pose risks to C-based open-source software. Our research offers practical solutions to minimize the threat of large-scale attacks on such software. Our literature review aimed to present a thorough overview of the existing research on the application of model checking for the identification and verification of vulnerabilities in open-source software. Although substantial work has been undertaken in this domain, our findings suggest a continued need for research, especially in developing tools capable of reducing the likelihood of software exploits. This underscores the ongoing challenge and necessity for advancements in security measures within open-source software. While various studies have been conducted, there's a clear need for more research, especially in creating tools to reduce software exploit risks.

Table 2 compares the tools described in this section when checking vulnerabilities in open-source software.

Tools	Year	BMC	Open-Source	Analysis Report
Chucky (YAMAGUCHI et al., 2013)	2013	✗	✓	✓
VeriFast (PHILIPPAERTS et al., 2014)	2014	✓	✗	✓
CBMC (KROENING; TAUTSCHNIG, 2014)	2014	✓	✓	✗
NuXmv (CAVADA et al., 2014)	2014	✓	✗	✗
SMACK (RAKAMARIĆ; EMMI, 2014)	2014	✓	✗	✗
Lazy-CSeq (INVERSO et al., 2015)	2015	✓	✗	✗
Dsverifier (ISMAIL et al., 2015)	2015	✓	✗	✓
EMCDM (PIRA; RAFE; NIKANJAM, 2016)	2016	✓	✗	✗
MCMAS (LOMUSCIO; QU; RAIMONDI, 2017)	2017	✓	✗	✗
DIVINE (BARANOVÁ et al., 2017)	2017	✓	✗	✓
Vanguard (SITU et al., 2018)	2018	✗	✓	✓
Heaphopper (ECKERT et al., 2018)	2018	✓	✗	✗
CRed (YAN et al., 2018)	2018	✓	✗	✗
UPPAAL (GERKING; SCHUBERT; BODDEN, 2018)	2018	✓	✗	✗
DiVM (ROČKAI et al., 2018)	2018	✓	✗	✗
PeSCo (RICHTER; WEHRHEIM, 2019)	2019	✓	✗	✗
QASan (FIORALDI; D'ELIA; QUERZONI, 2020)	2020	✗	✓	✗
IC3 (LANGE et al., 2020)	2020	✓	✗	✓
QPR Verify (BÜNING; SINZ; FARAGÓ, 2020)	2020	✓	✗	✗
DARTAGNAN (LEÓN et al., 2020)	2020	✓	✗	✗
CPAchecker (BEYER; KEREMOGLU, 2011)	2020	✓	✓	✗
C-SMC (CHENOY et al., 2021)	2021	✓	✗	✗
UAFSan (GUI; SONG; HUANG, 2021)	2021	✗	✓	✗
Pono (MANN et al., 2021)	2021	✓	✗	✗
ESBMC (GADELHA; MENEZES; CORDEIRO, 2021)	2021	✓	✓	✗
Deagle (HE; SUN; FAN, 2022)	2022	✓	✗	✗
CBMC-SSM (FISCHER et al., 2022)	2022	✓	✗	✗
HEAPSTER (GRITTI et al., 2022)	2022	✓	✗	✗
2LS (MALÍK et al., 2023)	2023	✓	✗	✗
LF-checker (GERHOLD; HARTMANNS, 2023)	2023	✓	✗	✗
LSVerifier (SOUSA et al., 2023)	2023	✓	✓	✓

Table 2 – Main tools used to check software vulnerabilities using model checking techniques and their applicability on open-source software

Table 2 reviews the tools developed and actively supported in the past decade. Researchers have projected automated source code security scanning tools that can be used to scan C open-source code. It provides clear and actionable feedback over the last decade to help developers quickly identify and fix security defects in their code. Nowadays, model-checking techniques have been proven to be more precise and are thus more widely used. The use of model-checking techniques to check software vulnerabilities and their applicability to open-source software is growing exponentially. Model checking automatically verifies software and hardware designs for correct behavior, generally to avoid design errors, oversights, and vulnerabilities. Therefore, this table focuses on discussing the advantages of utilizing model-checking techniques to identify vulnerabilities in open-source software from the perspective of cost-efficiency due to the transparency of the code, as well as its possible scalability, allowing for verification to be done on a system-wide scale rather than for a small portion.

LSVerifier employs a static code analysis approach based on the BMC technique that operates independently of executions, enabling the verification of multiple properties simultaneously. This method is particularly effective in evaluating critical projects due to its comprehensive analysis capability. Unlike other methodologies and tools presented in Table 2, LSVerifier stands out by efficiently checking for code property violations and generating reports based on BMC counterexamples that can help software developers find issues in open-source projects as a basis for distinguishing between false positives and true positives. This feature is especially valuable in assessing large open-source software systems, where the scale and complexity of the code can present significant challenges. The application of LSVerifier to projects comprising millions of lines of code demonstrates its capability to handle extensive and complex software environments, making it a crucial tool for ensuring the integrity and security of significant software projects.

3.2 Formal Verification Tools for Analyzing Open-Source Software Security Vulnerabilities

The C programming language is extensively utilized for developing crucial software applications. Nonetheless, it does not offer built-in protection mechanisms, thereby placing the responsibility of memory and resource management squarely on the shoulders of developers. Failures or oversights in these areas can lead to unpredictable program behavior and security vulnerabilities. In response to these challenges, numerous studies have focused on employing automatic tools to verify safety properties in C programs, aiming to mitigate the risks associated with manual memory management and enhance overall software security (CORDEIRO; FISCHER, 2011; ROCHA et al., 2020). However, due to verification complexity and applicability, not all memory safety violations can be covered efficiently using such tools. To solve this problem, there are solutions for program testing with publicly available frameworks for static techniques, symbolic execution (BALDONI et al., 2018), dynamic approaches using fuzzing and sanitization (DINESH et al., 2020), abstract interpretation (RIVAL; YI, 2020), and BMC technique

(CLARKE; KROENING; LERDA, 2004a; Cordeiro; Fischer; Marques-Silva, 2012), for instance. Thereby, currently, software developers have many multifaceted solutions for program testing using static techniques by upstream tools that allow users to customize key parameters specific to the desired test scenario during test exploitation (SITU et al., 2018; FIORALDI; D'ELIA; QUERZONI, 2020; GUI; SONG; HUANG, 2021).

Vorobyov, Kosmatov, and Signoles (VOROBYOV; KOSMATOV; SIGNOLES, 2018) seek to assess state-of-the-art techniques, thus analyzing the performance of different automatic vulnerability detection tools for C programs. For this purpose, a database containing approximately 700 test cases representing security-related vulnerabilities, previously classified regarding memory safety, was used. The respective results indicate that verification tools provide adequate support for detecting problems arising from improper use of memory and undefined behaviors, thus attesting to their reliability in improving C code. However, the same authors limited their analysis to known test cases. In contrast, tools like LSVerifier perform static code analysis, which does not depend on executions and can verify multiple properties in a single run. This approach proves to be more effective when evaluating critical projects.

Another approach to verify security vulnerabilities in software is using fuzzing technique (BÖHME et al., 2017). In this matter, the authors in (ROCHA et al., 2020) present Map2Check, a software verification tool that uses fuzzing techniques, symbolic execution, and inductive invariants to check safety properties in C programs. Furthermore, it uses the LLVM compiler's infrastructure to instrument source code and monitor data from a program's execution. It then uses an iterative deepening approach using fuzzes and symbolic execution engines to check such properties (MENEZES et al., 2018; ROCHA et al., 2020). However, even though their experimental results show that Map2Check can be helpful to verify pointer safety-related properties, it has only been evaluated under SV-Comp benchmarks and not in the context of large software systems. In addition, its evaluation is currently limited to SV-COMP's benchmarks. In contrast, our work presents results for many large open-source practical software systems.

Nie, Jiang, and Ma (NIE; JIANG; MA, 2020) introduced an efficient computation tree logic (CTL) symbolic model-checking algorithm based on fuzzy logic, which ad-

dresses the state space explosion problem. Unlike conventional algorithms based on binary decision diagrams (BDDs), the proposed algorithm uses fuzzy logic to reduce the complexity of BDD computations by representing CTL formulas as fuzzy sets. It enhances scalability and efficiency and inherently supports behaviors with probabilistic and temporal constraints. However, it also demonstrated limitations in representing counterexamples. Considering LSVerifier, we overcame this limitation by employing a model checker that generates counterexamples when properties are satisfied.

Alshmrany *et al.* (ALSHMRANY et al., 2021) introduced FuSeBMC, a method that combines fuzzing and BMC to find security vulnerabilities in C programs. This tool is based on ESBMC, providing an efficient test generation framework. They have demonstrated the effectiveness of their approach by using it to detect SQL injection bugs in a sample web application implemented in the C language. Given that this tool introduces a fuzzing component for code analysis, this condition may result in longer verification times and extra configuration effort compared to LSVerifier, which employs only BMC and still yields satisfactory results. The waiting time for results can be a critical factor in large projects.

Aljaafari *et al.* (ALJAAFARI et al., 2022) introduced Ensembles of Bounded Model Checking with Fuzzing (EBF). It is a method that combines BMC and Gray-Box Fuzzing (GBF) to discover software vulnerabilities in concurrent programs. The resulting tool was capable of producing up to 14.9% more correct verification witnesses compared to using BMC tools alone. Furthermore, this tool successfully detected a data race bug in the open-source project wolfMqtt. It was run over the benchmarks used in SV-COMP 2022. However, regarding practical software, its evaluation was limited to wolfMqtt and three other programs. In contrast, LSverifier was evaluated against various open-source large projects. Considering the performance gain achieved by adding fuzzing, it may be interesting to include this technique in future LSVerifier versions as there is potential to discover more bugs in open-source projects.

Richardson (RICHARDSON, 2020) describes using CHERI to provide memory safety in C/C++ programs. In particular, they show how to overcome memory errors, such as spatial safety violations, because the memory bounds of an object are ignored.

Moreover, the author presents CHERI sub-object hardening (CheriSH), a technique that protects against buffer overflows between the same object fields, enabling complete spatial memory protection in CHERI (RICHARDSON, 2020). Nonetheless, CHERI should be adopted by developers and industries so that processor architectures can benefit from such a mechanism. CheriSH primarily focuses on pointer safety, not tackling other violations, e.g., memory leaks. This method does not address all the security aspects that LSVerifier does and has not been tested on large open-source software systems (BRAUSSE et al., 2022).

Regarding tool properties, the available literature shows that BMC has already been applied successfully to discover flaws in real systems and has also been extended to support multi-threaded software systems (ROCHA et al., 2012; Barreto; Cordeiro; Fischer, 2011). Cordeiro, Fischer and Marques-Silva (Cordeiro; Fischer; Marques-Silva, 2012) investigate SMT-based verification of ANSI-C programs, focusing on embedded software, thereby offering the first SMT-based BMC assessment in industrial applications. The results reported there conclude that the ESBMC outperforms CBMC and SMT-CBMC when considering the verification of embedded software. Other studies have sought to extend or improve existing tools, such as those described in (BOUDJEMA et al., 2018) and (Cho; D'Silva; Song, 2013). However, this approach does not cover other security aspects addressed by LSVerifier, nor has it been applied to large open-source software systems.

Using another model-checking technique, the research realized by Gerking *et al.* (GERKING; SCHUBERT; BODDEN, 2018) describes the construction of a tool using the well-established Uppaal model checker to realize test automata. It introduces a dedicated location to identify violations of noninterference whenever it is reachable during execution. Therefore, this tool can reduce the problem to a reachability test supported by model-checking techniques used in software engineering practice. However, it was not applied to open-source software. However, it does not handle other security aspects addressed by LSVerifier and was not applied to large open-source software systems.

In addition, we can mention LLBMC (MERZ; FALKE; SINZ, 2012b) and DIVINE (BARANOVÁ et al., 2017). Both use BMC techniques to verify memory safety

properties. LLBMC is an interesting bounded model checker based on SMT solvers; however, it is limited to bounded analysis and program-dependent restricting tool scalability for large systems. DIVINE, an explicit-state model checker, is an efficient and versatile tool for analyzing real-world C and C++ programs. It provides a modular platform for the verification of real-world programs. However, a recent study (MONTEIRO; GADELHA; CORDEIRO, 2022) performed an extensive evaluation and found that DIVINE needs improvement regarding performance and reliability. LSVerifier does not present such limitations. It has already been applied to large software systems, as shown here, and its reliability finds support in the ESBMC's results obtained in many SV-COMP editions.

Choi(CHOI, 2011) introduced a model-checking technique to identify subtle problems in software safety applied to open-source software. This research reported a complete experience report with the Trampoline OS software safety analysis using the model checker SPIN. Trampoline OS is an open-source automotive electronic/electrical device operating system based on the OSEK/VDX international standard. The authors converted the Trampoline kernel code into formal models and experiments using an incremental verification approach. Furthermore, the automated counterexample generation guaranteed a useful tool for tracing potential safety bugs. This study was the first successful research case that used model-checking techniques to verify vulnerabilities in open-source software, indicating a trend in using model-checking tools in large-scale projects. LSVerifier, in turn, directly checks source code for a broad set of possible vulnerabilities without explicit manual conversion. In addition, Trampoline has only 4530 code lines, and no scalability assessment was performed. In that sense, LSVerifier was applied to projects with millions of code lines.

To analyze vulnerabilities in heap implementation, Moritz Eckert et al (ECKERT et al., 2018) proposed a tool called HEAPHOPPER. This is a novel and fully automated tool using model checking technique and symbolic execution, to analyze the exploitability of heap implementations in open-source. HEAPHOPPER tool has good results exploiting memory library allocation implementation in the presence of memory corruption. However, this work is limited to checking only memory allocation issues, which

does not happen with LSVerifier. It can not handle large software systems, which is the focus of our work. The authors also mentioned the need to enhance HEAPHOPPER's performance as the number of paths to be analyzed inevitably grows.

Another approach to verify software vulnerabilities is the code browser technique. Cobra (HOLZMANN, 2017) uses a lexical analyzer to scan source code and create an uncomplicated linked list of lexical tokens. Structural code analysis and pattern identification can be assisted by this tool. Patterns for true positive and false positive cases are carefully defined for every syntax rule or recommendation. If the code context matches a true positive pattern, the warning is considered a true positive. If the code context matches a false positive pattern, the checked warning is deemed a false positive. It will be classified as unknown without a pattern matching in the code context. The study by Thu-Trang *et al.* (NGUYEN et al., 2019) showed that Cobra can identify both true positives and false positives for rules and recommendations about program syntax. LSVerifier has a more efficient methodology for checking code property violations than this method by producing counterexamples as a standard for determining whether property violations are false positives or true positives.

Recently, Cook *et al.* (COOK et al., 2020) presented the use of model checkers to triage the severity of security bugs in the cloud service provider at Amazon Web Services (AWS). The authors tackled the severity of bugs discovered/reported in the Xen hypervisor, an open-source hypervisor used in industry. In this case study, when a bug is reported, engineers should evaluate its potential threat and how quickly it needs to be fixed w.r.t. its severity. To do so, the authors have applied transformations to the original source code and implemented modifications to the C Bounded Model Checker (CBMC) (CLARKE; KROENING; LERDA, 2004a), aiming to slice the program under verification and generate a reduced version of it. As a result, this model checker can easily verify the resulting program, while the obtained counterexamples can help engineers write security tests to analyze bugs further. However, several abstractions performed in the verification approach might cause the model checker to miss traces and not automatically falsify spurious traces. In this regard, it may be worthwhile to consider using techniques to simplify the programs analyzed by LSVerifier, making

them easier to verify. Moreover, it is worth mentioning that vulnerability severity is inherently considered in the LSVerifier's prioritization strategy, which can also be modified if new severity classes or a different ranking logic must be included.

Despite the surge in formal verification research to enhance model-checking performance for memory safety properties, most cutting-edge model-checking tools are tailored for analyzing small to medium-sized programs. A significant limitation of these tools is their often user-unfriendly outputs and tendency to generate false positives, particularly when functions undergo redundant analyses. In response, our work introduces structured log files and spreadsheets to streamline the organization of analysis results. We have also implemented a prioritization algorithm to help users systematically identify the most critical issues first. Our study seeks to bridge the gaps in verifying large software systems by developing a methodology specifically designed for this purpose. We aim to narrow the divide between research and practical application by employing a tool, LSVerifier, to evaluate real open-source software based on this methodology. Notably, LSVerifier distinguishes itself as the novel tool capable of producing reports derived from BMC counterexamples, thereby aiding software developers in identifying problems within open-source projects.

3.3 Summary

This literature review delves into the security vulnerabilities of C-based open-source software and the role of Bounded Model Checking (BMC) tools in verifying these vulnerabilities. The study is comprehensive, focusing on research from 2013 to 2023, and includes a systematic methodology for selecting relevant studies from prominent databases like IEEE, ACM, Science Direct, Springer, Scopus, Web of Science, and Google Scholar. The review uncovers 30 key papers, highlighting the evolution and application of various BMC tools like CBMC, VeriFast, NuXmv, and others. It emphasizes the increasing precision and popularity of model-checking techniques in open-source software verification, noting their cost-efficiency and scalability for large systems.

The second part of the review provides a detailed comparison of various tools

used in checking vulnerabilities in open-source software, emphasizing the advancements and limitations of each. Special attention is given to LSVerifier, a tool that uses BMC for static code analysis and can verify multiple properties in a single run. The review explores various approaches like fuzzing, symbolic execution, and static analysis, highlighting their effectiveness in detecting security vulnerabilities. Tools such as FuSeBMC, EBF, and CHERI are discussed, underscoring the necessity for efficient and reliable tools in large-scale software systems and the potential of combining different techniques like fuzzing with BMC.

The studies above show the importance of pursuing verification techniques for open-source software due to its intrinsic characteristics: collaborative development and code disclosure. It encompasses problems caused by the lack of knowledge during software implementation, given that third-party open-source libraries, components, utilities, and other open-source software are used in a bundle without further analysis. Indeed, most efforts focus on new techniques, and only a few initiatives try to efficiently or effectively tackle the massive amount of associated source code in open-source projects.

Finally, the review discusses the broader implications of these tools in the context of open-source software verification. It underscores the challenges of model-checking tools, such as handling large software systems and producing user-friendly outputs with minimal false positives. The study highlights LSVerifier's unique ability to generate actionable reports based on BMC counterexamples, aiding developers in identifying issues in open-source projects. The review concludes that despite significant advancements in formal verification and model-checking tools, there remains a gap in verifying large software systems efficiently, which LSVerifier aims to address. Due to this clear gap, this work will investigate and tackle security vulnerabilities in large C open-source code bases using a novel methodology that provides an automatic verification framework.

4

The Proposed Verification Methodology

This section introduces the proposed verification methodology for large software systems and the foundational architecture and key features of LSVerifier. The software organization that initially inspired its development will be detailed in the subsequent section, along with the core principles and concepts underpinning the implementation approach.

4.1 Architectural Patterns of Open-Source Software for Effective Vulnerability Verification

The proposed methodology targets the verification of large software systems, focusing on open-source software due to its prevalence and susceptibility to vulnerabilities. It is essential to recognize that large open-source software typically consists of a complex arrangement involving numerous files, diverse folder structures, and occasionally external repositories. For instance, notable examples such as PuTTY (PUTTY, 1999) comprise 175 files, OpenSSH (OPENSSH, 1999) includes 276 files, and OpenSSL (OPENSSL, 1998) contains 1,239 files. These systems are frequently integrated into many new projects for their crucial communication services, utilizing code that applies across various domains. The number of files in such software often correlates with its complexity and

directly reflects the chosen design strategy. Figure 5 shows the representation of the structural organization in large open-source software systems for security vulnerability verification.

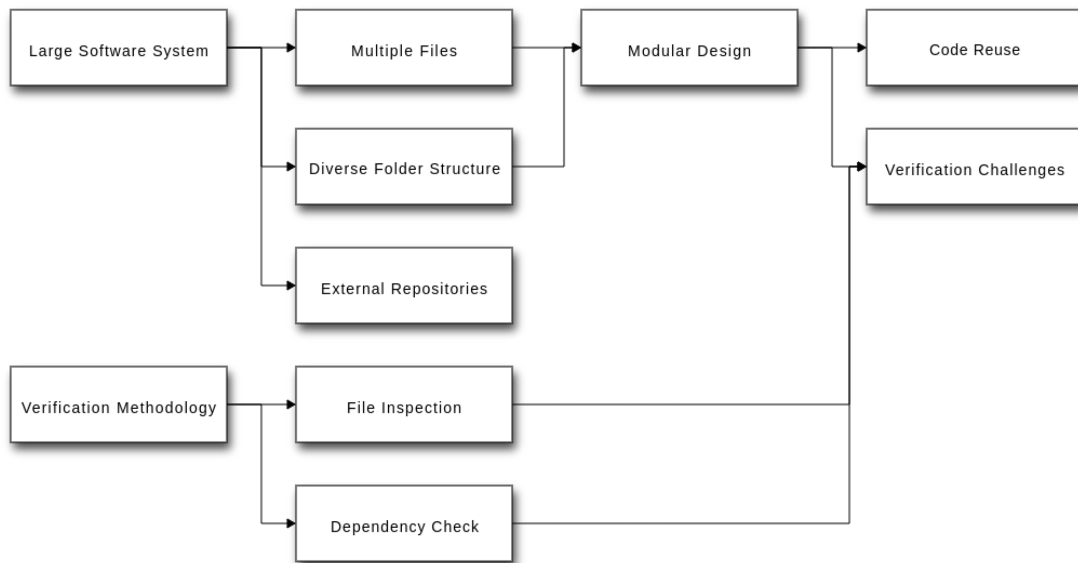

Figure 5 – Structural complexities in Open-Source Software for security verification

Dividing software into multiple files offers several advantages. It ensures that each file is manageable for easier editing, leading to improved organization and streamlined maintenance. Different software layers might be organized into separate folders, while distinct components are distributed across various files. This modular design facilitates code reuse in other software implementations, such as communication buses and network infrastructure. However, this type of organization also challenges the verification process: each file requires thorough inspection, and dependencies must be carefully considered when evaluating applications comprised of multiple components. Additionally, some files may not contain the 'main' function, which is often used as an entry point by software model checkers.

These factors were pivotal in shaping the development of our verification methodology for large software systems, forming the core requirements for any viable verification scheme.

4.2 The Development of Proposed Methodology

We have developed a new methodology based on BMC and prioritized search, whose general idea is shown in Figure 6. It guides an underlying model checker to verify a C program's entire code and can even reach third-party libraries. BMC was chosen as the underlying verification framework due to its performance and flexibility. It can ultimately provide a trade-off between effort (e.g., explored state space and resources) and effectiveness.

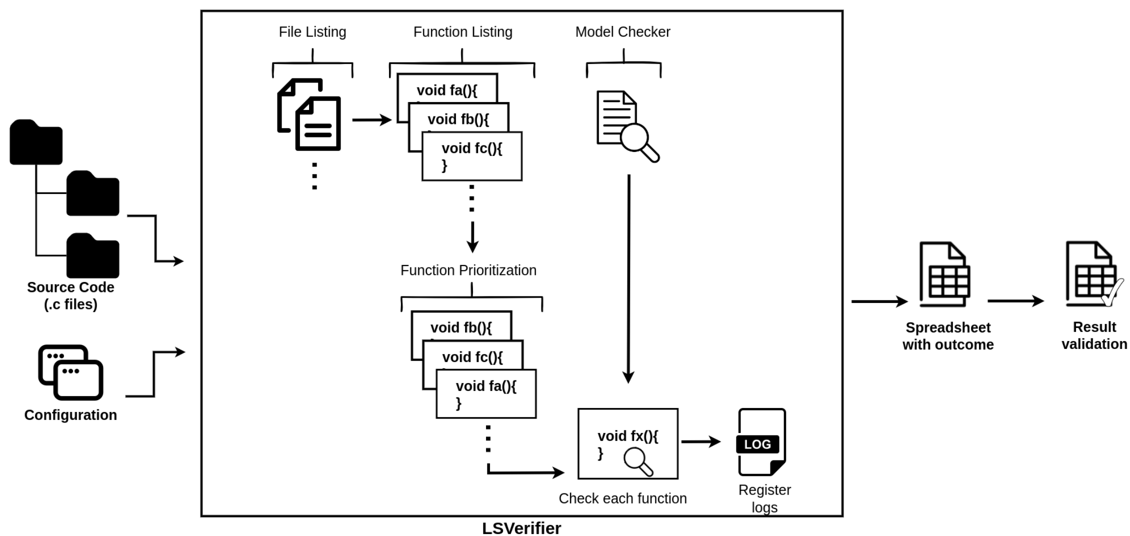

Figure 6 – An overview of the proposed verification process.

First, as shown in Figure 6, the target source-code directory and the necessary configuration, e.g., solver, encoding, and verification methods, are fed. Such information is paramount to match the characteristics of the code to be verified and the goal of the verification process itself. For instance, one might be interested only in a specific type of vulnerability, e.g., overflow. Next, all "*.c*" files and all their respective functions and methods are listed. Indeed, this is an important step that raises the potential locations for vulnerabilities.

However, in a verification process, one is undoubtedly interested in finding the most dangerous vulnerabilities first, which is inherently linked with the structure of a function or method, e.g., its signature. For instance, it is common sense in the programming community that elements taking raw pointers as parameters are likely more prone to errors than those not using them. Consequently, prioritization is also

performed, which can be used to drive further test and correction phases to handle more significant problems first. Next, an underlying verifier checks each function according to its priority and the initial configuration. This way, we ensure that the entire source code is evaluated in a prioritized fashion.

Finally, the associated verification logs are gathered and processed to create a spreadsheet with the related outcomes. Information regarding how files are verified, e.g., way of access and specific vulnerabilities, and how outcomes are displayed, e.g., amount of details, could also be fed.

A formal description of the proposed method, which can be used for the development of a real implementation of it, is given in Algorithm 1 and is explained as follows. Given a program P, in the directory D, and the configuration C, it first parses the latter. Then it lists the "$.c$" files of interest in D, which can be performed recursively or be focused on a specific element. As a result, the target files and the input configuration are stored for use in the following steps.

Suppose a user configures it to analyze program functions individually. In that case, each source-code file is checked, and a global list F of all functions and methods declared in them is generated, analyzed, and reorganized according to a prioritization based on its elements' structures. Consequently, elements are sorted from the most to the least priority. Otherwise, the analysis occurs based on the normal program flow, using the main function as a starting point with no prioritization scheme.

Following that, an underlying BMC checker analyzes each element in F, verifying violations such as pointer safety, arithmetic overflow, division by zero, and out-of-bounds arrays. Then, a suitable module generates the respective logs. In that sense, properties to be checked are passed to a controlling script in the argument $configs$. When the associated verification process is concluded, a spreadsheet V with the model checker's outcomes (complete report) is produced, and the compound parsed result of all logs is obtained from a complete execution process.

It is worth noting that our methodology operates file-wise, which is appropriate and necessary. In other words, a file's content, e.g., a complete element, service layer, or interface, is completely verified. This way, we can validate source code by investigating

building blocks and thus clearing them one by one.

Algorithm 1 The proposed verification approach.

Require: Program P, Directory D, Configuration C
Ensure: Verification Outcome V
  $configs \leftarrow get_configs(C)$
  $function_analysis \leftarrow extract_config(configs)$
  $files \leftarrow list_files(P, D)$
  $num_files \leftarrow length(files)$
  $log \leftarrow \emptyset$
  $k, l \leftarrow 1$
  while $k \leq num_files$ **do**
    if $function_analysis$ **then**
      $F \leftarrow list_functions(files[k])$
      $F \leftarrow prioritized_functions_list(F)$
    else
      $F \leftarrow main_function$
    end if
    $num_functions \leftarrow length(F)$
    while $l \leq num_functions$ **do**
      $log \leftarrow log \cup BMC_Check(files[k], F[l], configs)$
      $l \leftarrow l + 1$
    end while
    $k \leftarrow k + 1$
  end while
  $V \leftarrow spreedsheat_create(log)$
  return V

At this point, it is important to clarify the prioritization strategy for functions and methods, i.e., the function *prioritized_functions_list* in Algorithm 1. Indeed, there can be elements with different signatures, including return and parameter types, which give clues regarding their likelihood to present severe errors. Consequently, depending on them, some components should be evaluated first. For instance, the ones that have pointers or arrays as parameters present an inherent priority. Pointers may be wrongly used throughout a given piece of code and cause many problems due to direct memory manipulation. At the same time, arrays may suffer from improper access and incorrect use of their parameters.

Moreover, analyzing their bodies is also important as it complements the search for potential vulnerabilities. In this context, we first check for dynamic memory allocation, which can potentially cause memory leaks if a developer does not deallocate memory blocks properly. Specifically, we look for the use of functions malloc and free

to identify this scenario. Next, still within function bodies, we check for the use of asynchronous processing through threads, which has the potential to lead to concurrency issues such as race conditions and deadlocks. In this case, we verify the use of POSIX threads through functions such as *pthread_create* and *pthread_join* (BARNEY, 2009). Finally, the proposed prioritization strategy searches for arithmetic and logical operations, such as division and bitwise shifting. These may lead to overflow and undefined behaviors due to how compilers handle them and convert them into low-level instructions.

Consequently, functions are ranked according to a numerical prioritization scale ranging from grade 5 (the highest priority) to grade 0 (the lowest priority), based on their contents:

- the presence of pointers as parameters present the maximum priority and leads to grade 5;
- the use of arrays as parameters indicates grade 4;
- when dynamic memory allocation is present in a function's body, we tag it with grade 3;
- thread manipulation code results in grade 2;
- functions containing arithmetic operations or bit shifting are classified as grade 1;
- the remaining ones are considered low-priority functions and are then tagged with grade 0.

Regarding the prioritization algorithm, it is important to highlight an interesting point: our methodology also needs to examine the body of each function to look for other functions called within it. Indeed, if there is a call to another function already present in the list of elements to be analyzed, this is removed from it. It is necessary because some elements can be verified more than once since BMC checkers usually follow a function's flow and automatically analyze all elements along it without the need for an explicit request. Consequently, such a removal step inherently reduces execution times by avoiding redundant analysis procedures.

Algorithm 2 gives a formal description of the function *prioritized_functions_list* used in Algorithm 1. It first reads the initial list of functions F and then removes from it functions already called inside other elements ($remove_from_list$). Then, if the current function is not part of another context, it assigns different grades for each one, which is done according to the earlier prioritization scheme. Next, a new list F^o is assembled and returned, which is done with the function $sort_functions_by_priority$. Its content is sorted in descending order based on the newly assigned priorities.

Algorithm 2 The proposed prioritization algorithm.

Require: Function List F
Ensure: Ordered Function List F^o
  $num_functions \leftarrow length(F)$
  $k \leftarrow 1$
  while $k \leq num_functions$ **do**
    if $\forall f \in F : F[k] \supset f$ **then**
      $remove_from_list(F, f)$
    else if $F[k] \supset pointers$ **then**
      $F[k].priority \leftarrow$ grade 5
    else if $F[k] \supset array$ **then**
      $F[k].priority \leftarrow$ grade 4
    else if $F[k] \supset (malloc \mid free)$ **then**
      $F[k].priority \leftarrow$ grade 3
    else if $F[k] \supset threads$ **then**
      $F[k].priority \leftarrow$ grade 2
    else if $F[k] \supset (arithmetic\ operations \mid bit - shift\ operations)$ **then**
      $F[k].priority \leftarrow$ grade 1
    else
      $F[k].priority \leftarrow$ grade 0
    end if
    $k \leftarrow k + 1$
  end while
  $F^o \leftarrow sort_functions_by_priority(F)$
  return F^o

The outcomes of a complete verification procedure are exported to a spreadsheet containing all property violations found by the underlying BMC checker. This report aims to provide a clear and concise overview of the identified vulnerabilities, including detailed information on each such as property, file name, function name, and code line where it was detected. This information allows developers to locate and investigate the specific code that may be causing a vulnerability and may also estimate its potential impact.

4.3 Architecture and Main Functionalities

As our methodology for verifying large software systems had been completely devised, the next logical step was its implementation as a real tool capable of being run and evaluated.

LSVerifier, which is the name chosen for this tool, as described in Algorithms 1 and 2, was implemented using the Python programming language (VANROSSUM, 1995). As the specific underlying BMC checker, ESBMC was chosen, which happened due to its performance in previous instances of the International Competition on Software Verification (SV-Comp) (BEYER, 2020; BEYER, 2022). LSVerifier supports all aspects of C11 (STANDARDIZATION, 2012), the current standard for the C programming language, and detects vulnerabilities in software by simulating a finite prefix of program execution with its possible inputs. Also, an input program is verified by explicitly exploiting interleavings, where one symbolic execution per interleaving is produced.

By default, LSVerifier can check a range of software vulnerabilities aligned with MITRE's "Top 25" CWE list (MITRE, 2023). Its detection suite encompasses but is not limited to out-of-bounds array access, inappropriate pointer dereferences, arithmetic underflows and overflows, and dynamic memory allocation problems. Furthermore, by ESBMC checker, LSVerifier can identify unique anomalies such as not a number (NaN) in floating-point computations, division by zero errors, and more complex issues like memory leaks. The tool also recognizes multithreaded concerns including data races, deadlock, and atomicity violations, offering a holistic approach to software vulnerability detection. To check software vulnerabilities, it is necessary to specify command-line options linked to the control core's and ESBMC's options. The vulnerability classes that can be detected include (according to Section 2):

- out-of-bounds array access;
- illegal pointer dereferences (null dereferencing, out-of-bounds dereferencing, double free, and misaligned memory access);
- arithmetic under and overflow;
- NaN occurrences in floating-point;

- division by zero;
- memory leak;
- dynamic memory allocation;
- data races;
- deadlock;
- atomicity violations at visible assignments.

In addition, LSVerifier can prioritize the chosen vulnerability classes according to the parameters configured in the command line. It is also possible to select specific vulnerabilities to be checked. Code exploitation can be prioritized based on specific function types in source code (see Algorithm 2). The following sections will describe, in detail, the most important implementation modules of the proposed methodology, including the ESBMC's operation and the LSVerifier's configuration.

4.3.1 The LSVerifier's Tool Implementation

The proposed tool was developed in Python version 3.8. It is released under the Apache License 2.0 open-source software.

The source code, documentation, usage instructions, and installation information are available in the LSVerifier's repository[1]. Also, Zenodo repository [2] provides all scripts, benchmarks, tools, and instructions to run tests. LSVerifier can be installed using the tool *pip* (Python package installer), with the Linux command in Listing 1.

listing 1 The Linux command to install LSVerifier

```
$ pip3 install lsverifier
```

The LSVerifier allows control via command-line options informed in Table 3. These options control the following processes: file listing, function verification, outcome

[1] <https://github.com/janislley/LSVerifier>
[2] <https://zenodo.org/records/10077388>

display, ESBMC's options, and pointer checking control. The latter refers to a flag responsible for disabling pointer checks during the execution of ESBMC.

Table 3 – The LSVerifier's Configuration Parameters

Parameter	Description
-h, –help	Shows the available options.
-e, –esbmc-parameter	Defines the parameters to be provided to ESBMC.
-l file	Provides a file with paths for including header files from dependencies.
-f, –function	Enables the function verification.
-fp, –function-prioritized	Enable Prioritized Functions Verification.
-v, –verbose	Enables the verbose mode.
-r, –recursive	Enables the recursive verification.
-d dir	Sets the directory to be verified.
-p	Specifies the vulnerability class to be checked.
-fl file	Specifies a single file to be verified.
-dp	Disables pointer verification.

LSVerifier methodically verifies various software vulnerabilities and offers customization options through **-e** and **-f** for exploring specific properties in C source code. Although initially designed for integration with ESBMC, LSVerifier can be adapted for use with other software model checkers by modifying its arguments to align with different verification parameters.

4.3.2 File Listing

File listing in LSVerifier is achievable in three modes: for a single file using **-fl**, for all elements in the current directory, or through a recursive search with the **-r** option. The single file method focuses on one specific file, typically examining its main function. The directory and recursive methods employ the *glob* module for UNIX-style pathname expansion, allowing for pattern-based file selection, such as $''*.c''$ for C source files.

4.3.3 Function Listing and Prioritization

In addition, file listing can be performed in three ways: a single file, all elements in the current directory, and a recursive search. The parameter **-fl** must be used to verify a

single file, informing which element will be handled. This way, it is possible to check all functions of that file or just the main one, the latter being the usual operation mode for most model checkers.

Currently, directory and recursive listing are performed with the option **-r**, using the *glob* module, i.e., the UNIX style path name expansion (FOUNDATION, 2021). It is employed to find a path name with a specific pattern following the Unix shell's rules. So, for instance, to list all source files written in C, the pattern "$*.c$" is used.

4.3.4 Exporting results

After verifying each function in the target files, LSVerifier generates a verification report bearing the corresponding verification outcome. It is encapsulated into a spreadsheet of type comma-separated values (CSV), which allows easy handling and use.

This concise report is employed to analyze an error, understand its root, and correct either the initial specification or the input software. The following items are present in every resulting spreadsheet report written by LSVerifier:

- filename;
- verification status (e.g., failed);
- function name in which the violation was found;
- line number in which the function was called;
- violation type (e.g., NULL Pointer).

If a user wants to check specific results, a companion log file is also provided, which gathers all outputs obtained during an execution.

4.4 An Illustrative Examples of Using LSVerifier

As an illustrative example, this section describes using LSVerifier in a real verification process executed for PuTTY (PUTTY, 1999), a popular network file transfer application.

To check a software piece with several files, as with PuTTY, we have to run it in the directory where its source code is located.

In C language programming, developers often employ header files that contain declarations of constants, macros, and functions. Compilers typically search for these resources in default directories and folders specified by the compilation command. In our case, their paths must be manually listed in a text file, typically named *dep.txt*, which is then passed to LSVerifier during its execution using the parameter **-l** so that it can map them (e.g., third-party libraries). Listing 2 illustrates an example of such a file for PuTTY, where each path is included, one per line.

listing 2 The header paths in file (dep.txt)

```
/usr/include/gtk-3.0/
/usr/include/glib-2.0/
/usr/include/pango-1.0/
/usr/include/cairo/
/usr/include/gdk-pixbuf-2.0/
/usr/include/atk-1.0/
            ...
```

After listing the associated dependencies, we can run LSVerifier using the parameters described in Section 4.3.1. The example in Listing 3 illustrates a verification process for an entire project, running LSVerifier with arguments configured according to our previous explanation, which was used for PuTTY.

listing 3 A command that allows the LSVerifier's function-by-function verification for an entire project

```
   $ lsverifier -v -r -f -l dep.txt
```

Figure 7 shows a counterexample for a division-by-zero, which informs file name, verification status, function name, code-line number, and the type of software security vulnerability found. This information is in a report file and saved in a directory "/output".

During its execution, LSVerifier checks the properties mentioned in the associated command line, whose progress is informed via logs in the same console. Figure 8 contains the output of LSVerifier for Putty, while Figure 9 illustrates the verification report ($.csv$) generated for the same analysis procedure.

```
Counterexample:

State 4 file pcache.c line 278 function numberOfCachePages thread 0
----------------------------------------------------
Violated property:
file pcache.c line 278 function numberOfCachePages
division by zero
(signed long int)(p->szPage + p->szExtra) != 0

VERIFICATION FAILED
```

Figure 7 – Example of counterexample created by LSVerifier.

```
ESBMC version 6.7.0 64-bit x86_64 linux
file terminal/bidi_gettype.c: Parsing
Converting
Generating GOTO Program
GOTO program creation time: 0.143s
GOTO program processing time: 0.002s
Starting Bounded Model Checking
Symex completed in: 0.000s (14 assignmen
ts)
Slicing time: 0.000s (removed 12 assignm
ents)
Generated 0 VCC(s), 0 remaining after si
mplification (2 assignments)
BMC program time: 0.000s

VERIFICATION SUCCESSFUL

########################################
[FILE]: terminal/bidi_gettype.c [TIME]:
0.1685793399810791
########################################

[OVERALL TIME]: 19525.74449110031

########################################
Summary:

Files Verified:  403
Functions Verified:  1243
Counterexamples:  401

Overall time: 19525.74s
Peak Memory Usage: 19.35MB
########################################

LSVerifier - Verification is complete.
```

Figure 8 – The LSVerifier's output during a verification procedure for PuTTY.

There are other options for verification. Moreover, LSVerifier can verify specific "*.c*" files as illustrated below. With this command, it will check all functions passed as an input argument, as described in Listing 4.

listing 4 A command that allows the LSVerifier's verify specific .c file

```
$ lsverifier -v -r -f -fl main.c
```

file_name	functionVerified	function_name	function_line	error
cmdline.c	cmdline_host_ok	cmdline_host_ok	112	assertion cmdline_tooltype & TOOLTYPE
ldisc.c	ldisc_echoedit_update	backend_ldisc_option_state	837	dereference failure: invalid pointer
ldisc.c	ldisc_enable_prompt_callback	ldisc_enable_prompt_callback	205	dereference failure: invalid pointer
ldisc.c	ldisc_free	backend_provide_ldisc	839	dereference failure: invalid pointer
x11disp.c	x11_free_display	x11_free_display	17	dereference failure: invalid pointer
logging.c	log_free	logfclose	197	dereference failure: invalid pointer
psftp.c	list_directory_from_sftp_print	list_directory_from_sftp_print	84	dereference failure: invalid pointer
psftp.c	sftp_senddata	backend_send	820	dereference failure: NULL pointer
psftp.c	sftp_sendbuffer	backend_sendbuffer	822	dereference failure: NULL pointer
pageant.c	pageant_listener_free	sk_close	239	dereference failure: invalid pointer
pageant.c	pageant_listener_got_socket	pageant_listener_got_socket	1944	dereference failure: invalid pointer
pageant.c	pageant_reencrypt_all	ssh_key_free	902	dereference failure: invalid pointer
timing.c	run_timers	run_timers	202	dereference failure: invalid pointer
callback.c	queue_idempotent_callback	queue_idempotent_callback	38	dereference failure: invalid pointer
sshpubk.c	key_type	fread	77	dereference failure: invalid pointer
sshpubk.c	ppk_encrypted_f	read_header	482	dereference failure: invalid pointer
sshpubk.c	rsa1_save_f	rsa1_save_sb	405	dereference failure: invalid pointer
sshpubk.c	ssh1_write_pubkey	ssh1_pubkey_str	1601	dereference failure: invalid pointer
settings.c	do_defaults	strcmp	92	dereference failure: invalid pointer
settings.c	get_sesslist	get_sesslist	1352	dereference failure: invalid pointer
cmdgen.c	main	main	289	dereference failure: array bounds violated

Figure 9 – The verification report generated by LSVerifier for PuTTY.

In addition, the tool can verify specific properties, as described in Listing 5.

listing 5 A command that allows the LSVerifier's verify specific properties

```
   $ lsverifier -v -r -f -p memory-leak-check,overflow-check,
deadlock-check,data-races-check
```

4.5 Summary

This section presents a comprehensive overview of the proposed methodology for verifying security vulnerabilities in large software systems, focusing on the architecture and functionalities of the LSVerifier tool implementation. It highlights the challenges and strategies for managing the complexities inherent in large-scale open-source software systems, which often consist of extensive file and folder structures, necessitating a meticulous verification process.

The methodology is built upon Bounded Model Checking (BMC) and incorporates a prioritized search mechanism to systematically evaluate the memory safety issues in C programs, including their dependencies on third-party libraries. This approach is designed to balance the verification effort against its effectiveness, ensuring a thorough examination of potential security vulnerabilities.

LSVerifier, the tool developed to implement this methodology, supports a comprehensive range of software vulnerabilities aligned with MITRE's "Top 25" Common Weakness Enumeration (CWE) list. Its capabilities extend to detecting various issues such as out-of-bounds array access, illegal pointer dereferences, arithmetic overflows, and dynamic memory allocation problems, among others. The tool prioritizes function verification, allowing for targeted analysis of code sections more likely to harbor critical vulnerabilities.

The tool's implementation, described in Python, leverages the ESBMC model checker for its underlying verification engine, supporting all aspects of the C11 standard. LSVerifier operates with flexibility, allowing users to specify a range of command-line options to tailor the verification process to their specific needs, including file listing, function verification, outcome display, and the selection of specific vulnerabilities for checking.

An illustrative example demonstrates the application of LSVerifier in verifying the PuTTY software, showcasing the tool's ability to handle real-world software systems. The verification process involves listing the necessary dependencies, executing the tool with appropriate configurations, and analyzing the generated counterexample report for identified potential vulnerabilities. The detailed architecture and functionalities of LSVerifier, combined with a practical example of its application, illustrate the effectiveness of the proposed methodology in addressing the challenges of verifying security vulnerabilities in large-scale software systems. The tool's design reflects a comprehensive approach to vulnerability detection, prioritization, and reporting, making it a valuable resource for developers and security analysts working with complex open-source software projects.

5

Experimental Evaluation

In this section, we evaluate our approach to verifying large software systems, specifically focusing on open-source projects in the C language. We first outline our chosen setup, followed by a definition of our experimental goals. We also provide resources and guidance on reproducing our experiments, encompassing scripts, benchmarks, tools, and instructions. Subsequently, we discuss our results, highlighting identified vulnerabilities, resource allocation, and the acknowledgment and rectification of issues. Lastly, we address potential threats to the validity of our experiments.

5.1 Experimental Setup

All experiments described in this work, using LSVerifier, were performed on a personal computer with an Intel(R) Core(R) i7 CPU 9750H processor and the Ubuntu 20.04 operating system. Moreover, it ran under a clock of 2.60 GHz and used 32 GB of random access memory (RAM).

All execution times presented here are CPU times, i.e., only the elapsed periods spent in the allocated CPUs, measured with the Linux tool *time* (PAGE, 2023). LSVerifier used this procedure to compute the total time consumed when verifying software vulnerabilities. Additionally, an approach was devised to assess the peak memory allocation during verification processes. It was achieved using the module *tracemalloc* (DEBUGGING; PROFILING, 2023), which traces the allocated memory blocks and allows efficient and real-time tracking of memory consumption.

ESBMC v6.7.0 was employed for the verification of C programs. It focused on code robustness regarding accurate pointer utilization, appropriate access to contiguous memory blocks, detection of values leading to variable overflow, and identification of division by zero.

To evaluate our verification methodology, focusing on large software systems, we selected twelve prominent open-source programs written in the C programming language: VLC (VLC, 2001), VIM (VIM, 1991), Tmux (TMUX, 2007), RUFUS (RUFUS, 2011), OpenSSH (OPENSSH, 1999), CMake (CMAKE, 2000), Netdata (NETDATA, 2006), Wireshark (WIRESHARK, 1998), OpenSSL (OPENSSL, 1998), PuTTY (PUTTY, 1999), SQLite (SQLITE, 2000), and Redis (REDIS, 2009). They are distributed under open-source licenses, such as the GNU General Public License (GPL), the Apache License, and the Massachusetts Institute of Technology (MIT) License, and more details for each can be found in the respective code repositories.

The selected programs for this study were chosen based on three key aspects: their substantial code size, significant importance to the open-source community, and extensive use of linked third-party libraries. Table 4 shows each open-source project and its respective software version used during experimental evaluation.

Software	**Version**
VLC	3.0.18
VIM	9.0.1672
TMUX	3.3a
RUFUS	4.1
OpenSSH	9.3
CMake	3.27.0-rc4
Netdata	1.40.1
Wireshark	4.0.6
OpenSSL	3.1.1
Putty	0.78
SQLite	3.42.0
Redis	7.0.11

Table 4 – Open-source software projects verified by LSVerifier.

5.2 Experimental Objectives

We have employed LSVerifier to verify the software modules listed in Table 5, where all C files were individually analyzed by checking each function. We have also used flags for log plotting and provision of the ESBMC's configuration.

EG1 **(Automation and Scalability)** Does our approach yield results in a reasonable timeframe and scale efficiently to handle large software systems without requiring manual intervention?

EG2 **(Practical Use and Effectiveness)** Is our methodology capable of identifying common issues that are corroborated by the developers of the respective software modules?

EG1 evaluates LSVerifier's capacity to guide its underlying checker, ESBMC, in adapting input code for efficient verification via automated reasoning. EG2, conversely, focuses on LSVerifier's application in practical software scenarios and its effectiveness in uncovering complex issues that might otherwise remain undetected.

5.3 Threats to the Validity of Experiments

In this section, we categorize the potential threats to the validity of our experiments into three distinct areas. This structured approach allows for a clear and focused examination of each category, enhancing the clarity and depth of our analysis.

5.3.1 Benchmark selection

Our methodology's assessment utilized a set of open-source C software project benchmarks to gauge effectiveness and efficiency. However, the scope of this dataset is limited to the context of this study, and its results may not extend to other benchmarks. Selecting representative benchmarks to understand a scenario's strengths and weaknesses is crucial. Additionally, it is necessary to recognize the limitations of our results and the

possibility of variation in different contexts.

5.3.2 Performance and correctness

Our strategy assumes that evaluating each program function can lead to accurate verification. However, the correctness of our approach might be compromised if the verification assumptions do not fully capture the program's behavior, particularly in scenarios involving complex parallel or concurrent programming. Additionally, the performance could be affected when dealing with benchmarks influenced by these factors, as the verification might need to account for all possible function interleavings, which can be both computationally intensive and time-consuming.

5.3.3 Counterexample validation

Validating counterexamples generated by LSVerifier is critical for verifying program correctness. However, this validation can be challenging, particularly with complex software or large projects. Therefore, we have undertaken rigorous testing and analysis to ensure the accuracy and reliability of these counterexamples. Additionally, it is important to conduct broader evaluations encompassing various scenarios and error conditions to uncover any unforeseen issues with our tool.

5.4 Experimental Results

Our study used LSVerifier to verify the software modules detailed in Table 5. This process involved individual analysis of all C files, focusing on examining each function. Additionally, we utilized various flags for log plotting and configuring ESBMC's settings during the verification process.

The verification process using LSVerifier revealed numerous property violations, predominantly concerning 'dereference failure' as outlined in Section 2.4. Table 5 presents this data, including module names and versions, the count of violated proper-

ties, the quantity of ".c" files, the number of external ".h" inclusions, source-code lines, verified functions, execution times, and peak memory usage. Key vulnerabilities identified in the analysis include pointer dereference, division by zero, dynamic object, and array-bounds violations, all crucial according to the Common Weakness Enumeration (CWE) standards.

Table 5 – Software vulnerabilities detected by LSVerifier, correlating the number of code-property violations with the amount of C code files, external libraries, lines of code, functions verified, maximum necessary memory, and verification time

Software Project	Software Version	Property Violated	Files	External Includes	Source-code Lines	Functions Verified	Verification Time	Memory Usage
VLC	3.0.18	72	1171	289	421840	13709	1033.79s	20.09MB
VIM	9.0.1672	110	188	95	366775	9611	554.56s	39.83MB
TMUX	3.3a	1788	179	445	61004	2168	52218.45s	43.12MB
RUFUS	4.1	576	144	108	56278	1615	283.95s	6.06MB
OpenSSH	9.3	338	290	63	109791	3183	873.27s	42.58MB
Cmake	3.27.0-rc4	552	1516	1030	324760	11279	934.21s	37.07MB
Netdata	1.40.1	1318	307	160	312530	7352	51471.27s	129.09MB
Wireshark	4.0.6	2141	2330	77	4177163	121567	59952.39s	391.44MB
OpenSSL	3.1.1	3140	1575	616	491632	17168	6046.63s	53.34MB
PuTTY	0.78	2472	403	153	127282	5310	66210.32s	58.54MB
SQLite	3.42.0	3265	340	609	258382	8911	2493.75s	33.22MB
Redis	7.0.11	187	418	556	170673	8211	727.76s	46.57MB

LSVerifier was able to detect potential vulnerabilities in every tested software project. The lowest number of violations occurred with VLC, which is not necessarily surprising, considering its development time (20 years) and its importance as one of the currently-available leading open-source media players. Although VLC has 13709 functions split throughout 1171 files, with 421840 lines of code, its verification time is relatively short (above 17 minutes) compared with others available in the same table. For instance, Tmux, which comprises 2168 functions distributed across 179 files, totaling 61004 lines of code, required approximately 14.5 hours for verification.

When applied to other large open-source software such as SQLite, OpenSSL, Putty, and Wireshark, all with more than 120000 code lines, LSVerifier identified many property violations: 3265, 3140, 2472, and 2141, respectively, as shown in Table 5. These high numbers of property violations are primarily a consequence of using multiple third-party library dependencies to whose implementations LSVerifier does not have access for verification. It leads to numerous header file inclusion (i.e., ".h" files) in source code: 609 for SQLite, 616 for OpenSSL, 153 for PuTTy, and 77 for Wireshark. Such behavior leads to problems not considered in unit tests or tackled during testing rounds.

The highest peak memory usage was observed with Wireshark, which presents a complex code organization and the highest number of functions, files, and lines of code. This project presents 77 header-file inclusions, as already mentioned, and undertakes the complex task of network traffic analysis. When used as a tool, running on Linux, Wireshark usually requires more than 500 MB of RAM, which already hints at its high resource demands. In addition, Wireshark imposes no limit on the number of packets it can handle, which creates a rich state space to be explored by LSVerifier. Netdata is another project that requires high memory usage for analysis, which performs media decoding and presentation. This project is organized in 312530 lines of code and includes 160 different external header files.

In terms of verification duration, PuTTY required the longest time, which was unexpected given its relatively small codebase, consisting of 127282 lines of code, 5310 functions, and 153 includes. Nevertheless, PuTTY executes complex operations involving encrypted communications and various protocols. Wireshark accounted for the second-longest verification time despite having a significantly larger code volume. This suggests that the analysis complexity is not solely dependent on the size of a program or the number of its header-file inclusions but rather on the intricacy of its programming and structures, which its dependencies may also influence.

LSVerifier was able to check programs with sizes ranging from 144 files, 1615 functions, and 56268 code lines, which is the case of RUFUS, to 2330 files, 121567 functions, and 4177163 code lines, which is the case of Wireshark. The obtained figures clearly show scalability capacity. Furthermore, the peak memory usage ranged from 6.06 to 391.44 MB of RAM, which is an acceptable amount given the typical hardware capabilities of modern personal computers. This indicates that LSVerifier can maintain low memory requirements, even with large software. It also significantly differs from other recent verification tools based on model checking, focusing primarily on execution speed and CPU usage (MANN et al., 2021; LANGE et al., 2020; CHENOY et al., 2021).

Putty required 18 hours to be verified, representing the longest verification procedure. In other words, software modules with hundreds of thousands or even millions of lines of code can be evaluated in less than one day in a completely automated

manner, which one can even trigger after a code-delivery meeting for nightly execution. Consequently, the above aspects reinforce the scalability properties of LSVerifier and integrally answer **EG1**: the proposed approach produces results in a reasonable period, and it can be applied to small and very large software systems indistinctly.

Although Table 5 shows the total number of violations, evaluating the prevalence of distinct vulnerability classes is also interesting. Table 6 presents the same verification results in Table 5 but now categorized into eleven different types of property vulnerabilities detected by LSVerifier, following the ESBMC's nomenclature:

- **Invalid Pointer (IP)**, which corresponds to null pointer dereferences, as described in Definition 3;
- **Array Bounds Violated (ABV)**, **Array Lower Bound (ALB)**, and **Array Upper Bound (AUB)**, which are specific cases where an array is accessed beyond its allocated boundaries, as detailed in Definition 8, and are intrinsically linked to buffer overflows, where data overruns the set limits of a buffer, as described in Definition 1;
- **Same Object Violation (SOV)**, which happens when pointers are compared in violation of the "same object" rule, i.e., C language allows the comparison of pointers using relational operators but imposes restrictions on their use when operands are pointers referring to the same address, as described in Definition 9;
- **Invalid Pointer Freed (IPF)**, which occurs when an uninitialized or invalid pointer is released using the function *free*, as described in Definition 9, and is similar but not the same as IP, focusing on memory deallocation;
- **Invalidated Dynamic Object (IDO)**, which corresponds to objects that become invalidated, often as a result of deallocation, can arise from various sources, mainly pointer problems, and is further elaborated in Definitions 3, 5 e 6;
- **Null Pointer (NP)**, which involves the inappropriate use of null pointers, typically by dereferencing them, as described in Definition 5, and is specifically about null pointers (IP and IPF may include other types of invalid pointers);

- **Division by Zero (DZ)**, which refers to cases where a number is divided by zero and causes undefined behavior, as described in Definition 7, and is also a specific type of arithmetic operation that can lead to undefined behavior, as described in Definition 2;

- **Assertion Failure (AF)**, which happens when a condition passed to the function *assert* is not met, indicating an error or unexpected behavior, as described in Definition 10;

- **Access to Object Out of Bounds (AOOB)**, which is a more general term that could refer to accessing any object (e.g., strings and linked lists), not just arrays, beyond their allocated boundaries, is primarily described in Definition 4, and is also associated with several other CWEs;

Table 6 – Software property violations found, using the chosen dataset, with LSVerifier and split into the 11 categories recognized by ESBMC

Software	Source	Includes	IP	ABV	ALB	AUB	SOV	IPF	IDO	NP	DZ	AF	AOOB
VLC	421840	9395	57	2	0	0	0	2	0	1	0	10	0
VIM	366775	443	100	3	1	2	0	2	0	2	0	0	0
TMUX	61004	1034	1725	0	12	9	0	21	0	20	0	1	0
RUFUS	56278	1453	513	0	0	4	4	6	0	20	29	0	0
OpenSSH	109791	3919	311	3	4	0	0	4	1	10	5	0	0
Cmake	324760	7710	481	28	5	2	18	7	0	6	2	3	0
Netdata	312530	1516	1045	1	12	5	3	5	0	212	35	0	0
Wireshark	4177163	19513	1940	20	12	17	35	2	0	77	5	27	6
OpenSSL	491632	9892	2753	77	98	22	10	7	2	131	11	29	0
Putty	127282	2041	1996	8	25	26	4	6	0	56	14	337	0
SQLite	258382	1224	2254	36	15	37	16	9	0	540	29	326	3
Redis	170673	2555	150	3	9	3	11	3	0	7	0	0	1

Some identified vulnerabilities were related to conditions that usually lead to memory corruption or crashes, e.g., accessing invalid pointers or out-of-bounds arrays. Although these definitions seem to present new and specific conditions, all of them can be traced back to the basic vulnerabilities in Section 2 as explicitly shown. The comprehensive list of CWEs supported by LSVerifier can be found in Table 7.

Table 7 maps the Common Weakness Enumeration (CWE) identifiers, as assigned by MITRE, to the types of vulnerabilities detectable by LSVerifier. This table showcases LSVerifier's comprehensive verification capabilities, especially considering that some vulnerability categories encompass numerous CWE identifiers. These vulnerabilities

underscore the importance of thorough software testing and validation conducted by LSVerifier. Employing tools for static analysis, dynamic analysis, and formal verification methods is essential in the early detection and mitigation of these risks, contributing significantly to software applications' robustness, reliability, and security.

Table 7 – The vulnerabilities identified by LSVerifier along with the corresponding CWE numbers for each property violation detected

Vulnerability type	CWE numbers
Invalid pointer (IP)	CWE-416, CWE-476, CWE-690, CWE-822, CWE-824, CWE-908
Array Bounds Violated (ABV), Array Lower Bound (ALB), Array Upper Bound (AUB)	CWE-20, CWE-119, CWE-120, CWE-121, CWE-125, CWE-129, CWE-131, CWE-193, CWE-628, CWE-676, CWE-754, CWE-755, CWE-787, CWE-788
Same Object Violation (SOV)	CWE-125, CWE-170, CWE-193, CWE-466, CWE-469, CWE-682, CWE-787
Invalid Pointer Freed	CWE-415, CWE-416, CWE-459, CWE-590, CWE-761, CWE-825
Invalidated Dynamic Object (IDO)	CWE-415, CWE-416, CWE-476, CWE-664, CWE-789
NULL pointer dereference	CWE-391, CWE-476
Division by zero	CWE-369
Assertion violation	CWE-190, CWE-191, CWE-389, CWE-478, CWE-571, CWE-569, CWE-617, CWE-670, CWE-680, CWE-681, CWE-682, CWE-685, CWE-754
Access to Object Out of Bounds (AOOB)	CWE-119, CWE-125, CWE-170, CWE-193, CWE-466, CWE-682, CWE-787, CWE-823

When analyzing Table 6, we can observe that the most prevalent vulnerabilities were the ones related to violations involving invalid pointers, i.e., IP and NP. Indeed, this sheds some light on memory corruption as a critical issue in C source code. Our results showed that overstepping bounds caused most identified pointer safety violations. In such scenarios, pointers were initialized with memory blocks allocated dynamically, and programming mistakes led to out-of-bounds errors. The highest number of pointer violations (IP and NP) was found in OpenSSL, which provides implementations for the protocols' secure sockets layer (SSL) and transport layer security (TSL), with basic

encryption capacity. Indeed, due to data-block encryption, there is a lot of memory allocation and pointer manipulation (e.g., using function pointers), which explains the behavior and the associated results.

OpenSSL presented the highest number of bounds violations, i.e., ABV, ALB, and AUB (197 occurrences). Again, it is also closely related to manipulating memory buffers, which are processed for subsequent use (BAGUELIN et al., 2022). Mishandling these buffers can lead to out-of-bounds errors when data is read or written beyond the allocated memory.

LSVerifier found 35 SOV and 6 AOOB occurrences in Wireshark. The latter deals with analyzing network packets, where pointers are extensively used to verify data fields in a unique comparison step, often involving strings and other objects. Consequently, it is expected that LSVerifier would detect operations comparing pointers that refer to the same address and access to objects.

Most IPF events were found in Tmux, a terminal multiplexer developed by various developers. Besides, it relies on external libraries while interacting with an operating system's APIs. During these operations, the resulting tasks undoubtedly require new memory blocks, which must later be freed. These complex routines can explain the obtained results.

SQLite is an engine for accessing structured query language (SQL) databases widely used in embedded device projects, which presented 540 errors in the NP category. Indeed, SQLite often deals with operation interleaving, corrupted databases, and statements, which may result in NP occurrences (SECURITY, 2022; SUCKEVIC, 2023). In other words, its routine tasks are closely related to this kind of fault, which should inherently lead to more careful coding and testing processes. Moreover, this aspect is challenging in the C language and requires expertise.

Netdata presented 35 DZ errors. It is a project designed to collect server metrics, assisting system administrators to take proactive measures. Due to its capability to retrieve statistics from different sources along with their associated data (e.g., timestamps), with subsequent computation involving these via external plugins, there is a high risk related to this kind of fault (FOUNTOULAKIS, 2020).

When analyzed by LSVerifier, PuTTY presented 337 AF occurrences. It supports network protocols and manages user inputs by applying concurrency through multiple threads. Consequently, issues related to data validation using assertions, which are usual in this task, can result in assertion failures.

It is worth noticing that some out-of-bounds errors occurred in scenarios where functions wrongly read data from heap-allocated memory. It can corrupt memory or induce unpredictable behavior. For instance, it is possible to exploit a bounds violation to write arbitrary code into specific positions of memory and execute that same code, which may result in losing control over the specific process, compromising the whole software module. OpenSSL, SQLite, and Putty presented the highest hits in the out-of-bounds category, with 197, 88, and 59 failures, respectively. These projects have a significant number of code lines that have been maintained and modified for a long time, which can explain their failures.

As a general comment, developers must be aware of potential memory management issues so they can take measures to prevent them, for instance, implementing defensive programming practices such as boundary checking on memory access operations. By prioritizing secure memory management practices, developers can help prevent serious software vulnerabilities in their projects.

Finally, we were able to find DZ vulnerabilities in VLC (LHOMME, 2022), RUFUS (OLIVEIRA, 2022f; OLIVEIRA, 2022j; OLIVEIRA, 2022d), OpenSSH (OLIVEIRA, 2022b; OLIVEIRA, 2022a), and Netdata (OLIVEIRA, 2022c; OLIVEIRA, 2022e). This kind of issue usually occurs when a parameter within a division operation determines the size of the variable to be created before executing an operation. Often, there is no check to assure that the divisor is strictly positive, resulting in an integer overflow bug and a division by zero. The highest threat from this vulnerability regards the system's integrity. Moreover, an attacker can easily disrupt its operability by sending an invalid interval value.

By leveraging LSVerifier's comprehensive detection capabilities, we can create a more robust and resilient software vulnerability management and prevention tool. The enhancement in security achieved through this approach mitigates risks and promotes

a more reliable, efficient, and secure software ecosystem.

In summary, LSVerifier can find usual problems in real software modules, which are widely known and understood by the development community. This aspect also leads to a prompt explanation and faster identification and correction. For instance, a developer can quickly understand an IP or NP occurrence, which is also mentioned in the LSVerifier's final report (spreadsheet). Consequently, this capability begins answering **EG2** because usual and easily understandable problems are found.

However, a deeper discussion regarding problem confirmation is still missing, which will be tackled in the next section.

5.5 Violated Properties Analysis

The full software verification process includes identifying vulnerabilities, confirming them, analyzing code, and implementing repairs, such as patch applications or merge requests. However, this comprehensive approach was not feasible for all programs listed in Section 5.1 due to constraints like infrastructure availability, time requirements, or developer availability. Nevertheless, we reported some discovered issues to VLC, VIM, RUFUS, OpenSSH, CMake, Netdata, Wireshark, OpenSSL, Putty, and Redis code repositories. These reports, submitted through specialized tools or email, confirmed vulnerabilities and subsequent application of remedial patches.

In this work, the issues reported were based on counterexample traces from LSVerifier and discussions with developers and maintainers, who either confirmed or labeled some as false positives. Table 8 details key property violations and bugs reported through GitHub or email. Despite the seemingly low number of issues in repositories, each requires significant time for registration, discussion, and resolution, often spanning months. Thus, we focused on reporting only the most critical ones, a decision supported by our prioritization strategy.

In our analysis of RUFUS, we identified property violations such as array bounds, division by zero, and invalid pointers. We logged three specific issues with the developers attributed to imported libraries. These issues are detailed in references

Table 8 – The reported issues involved code property violations and were submitted to the repositories of the respective software projects. These reports were made for the developers to confirm the associated software vulnerabilities

Software	Issues opened	Issues confirmed	Issues fixed
VLC	1 (LHOMME, 2022)	1	1
VIM	1 (OLIVEIRA, 2022k)	0	0
TMUX	1 (OLIVEIRA, 2023a)	0	0
RUFUS	2 (OLIVEIRA, 2022j; OLIVEIRA, 2022d)	2	1
OpenSSH	2 (OLIVEIRA, 2022b; OLIVEIRA, 2022a)	0	0
CMake	1 (OLIVEIRA, 2022g)	1	1
Netdata	2 (OLIVEIRA, 2022c; OLIVEIRA, 2022e)	0	0
Wireshark	1 (OLIVEIRA, 2022i)	1	1
OpenSSL	1 (OLIVEIRA, 2022h)	1	0
Putty	1 (E-mail)	0	0
SQLite	2 (OLIVEIRA, 2023b; OLIVEIRA, 2023c)	1	0
Redis	2 (OLIVEIRA, 2023e; OLIVEIRA, 2023d)	1	0

(OLIVEIRA, 2022f; OLIVEIRA, 2022j; OLIVEIRA, 2022d). Each issue highlights specific code errors and their implications, offering insight into the root causes and potential fixes for the identified vulnerabilities in RUFUS's software structure.

Thus far, we have got one bug fix for $tiny-regex-c$ to handle an out-of-bounds violation (OLIVEIRA, 2022j) related to CWE-787. In Listing 6, which shows real log information, one can notice that this problem happened in file *re.c*, one of the two that this dependence presents (the other is *re.h*). The value retrieved from the attribute *it type* of the object *pattern* can go beyond the maximum length of the array *types*.

listing 6 Bounds violation in tiny-regex-c used by Rufus

```
Building error trace

Counterexample:

State 5 file re.c line 269
In function re_print thread 0
----------------------------------------------
Violated property:
file re.c line 269 function re_print
array bounds violated:
array 'types' upper bound
(signed long int)(pattern +
(signed long int)i)->type < 17

VERIFICATION FAILED
```

To avoid such a condition, a specific check was added, shown in the code excerpt in Listing 7. It assures that the index passed to *types* is below its limit, i.e., *NOT_-WHITESPACE*, as declared in an enumeration. As one can see, this is a simple measure

that should always be adopted as common practice. However, it also reveals the careless coding performed by many developers.

listing 7 Corrected C program for tiny-regex-c

```
...
if (pattern[i].type <= NOT_WHITESPACE)
    printf("type: %s",
        types[pattern[i].type]);
else
    printf("invalid type: %d",
        pattern[i].type);
...
```

The second issue is a division by zero related to CWE-369. It was found in the library *ext2fs* and discussed with its developers (OLIVEIRA, 2022d). They acknowledged that if a *hashmap* with size 0 is created, it could cause a program crash. However, this is considered a bug at the application level based on the assumption that such an operation should not be performed. Its identification is shown in Listing 8.

listing 8 Divison by zero in ext2fs library used by RUFUS

```
Building error trace

Counterexample:

State 4 file hashmap.c line 51
In function ext2fs_hashmap_add thread 0
-----------------------------------------------
Violated property:
file hashmap.c line 51
in function ext2fs_hashmap_add
division by zero
h->size != 0

VERIFICATION FAILED
```

This argument holds from a purely functional standpoint. However, from a security perspective, any unexpected behavior, including crashes, should be considered a potential vulnerability. Treating such conditions as potential threats is necessary until they are comprehensively analyzed and discarded or properly registered. A check should be implemented to confirm that $h \rightarrow size$ is not 0 before the modulo operation, as shown in the code excerpt in Listing 9.

Again, this simple correction should be a coding rule and could prevent serious problems. In summary, we advocate that even if some vulnerability is not likely to

listing 9 Corrected C program for ext2fs-c

```
    ...
int ext2fs_hashmap_add(
    struct ext2fs_hashmap *h,
    void *data, const void *key,
    size_t key_len)
{
    // Check if h->size is zero
    if (h->size == 0) {
        // Handle the error
    }

    uint32_t hash =
        h->hash(key, key_len) % h->size;
    ...
```

happen due to a given program's structure, it must be handled. This way, even intended bad coding can be reduced, aiming at exploiting known vulnerabilities can be mitigated.

When checking the violated properties for VLC (LHOMME, 2022), some memory-safety vulnerabilities found during our experiments were reported via email. After their analyses, double-free errors were confirmed, a type of vulnerability related to CWE-415. This error occurs when software modules free a memory allocation twice, and doing so can lead to the modification of unexpected memory blocks, even resulting in a system crash or potentially allowing an attacker to execute arbitrary code. Specifically, this fix removed a deprecated Linux framebuffer (Kernel development community, 2023) plugin. Indeed, the Linux fbdev (KNORR; DANZER; UYTTERHOEVEN, 2023) subsystem has been deprecated for over a decade as better options are currently available.

We have also reported another issue caused by a third-party library in OpenSSL (OLIVEIRA, 2022h). The developers confirmed that an invalid pointer is dereferenced, which may likely cause a crash in the caller. Still, they do not consider this a vulnerability, as many OpenSSL APIs crash if a null pointer is passed to them. Here, we have a bad practice. Although a problem was found and confirmed, developers often state that a particular condition may never happen as a specific function or method is never invoked how it should be to provoke it. Suppose an attacker knows that and manipulates parameters or even codes to create that unfeasible scenario. In that case, the problem will happen and may even cause severe loss. Consequently, we also need to encourage behavior change, where any problem is handled properly and treated as a priority.

Anyway, such a condition should be monitored as it can be a source for an attack resulting in a system crash.

Regarding CMake, most of its problems are related to third-party libraries. Specifically, our analysis of its property violations revealed an important issue: a confirmed dereference failure caused by an invalid pointer. This error was fixed in the Cmake repository in function $cm_utf8_decode_character(...)$ and was caused by an empty input range local variable resulting in an invalid pointer (OLIVEIRA, 2022g), being related to CWE-824. Additionally, CMake employs third-party libraries such as $cmbzip2$ and $cmzstd$, each with its upstream software source, and our analysis uncovered issues in both. Specifically, we identified problems caused by invalid pointers, which lead to memory corruption and present an opportunity for arbitrary code execution. Furthermore, we could not locate any open-source repositories for $cmbzip2$ and $cmzstd$ where we could report these potential vulnerabilities. Here, we highlight another aspect: who is responsible for a given open-source module? Sometimes, it is difficult to answer such a question. Alternatively, it could be removed or even treated as a project's responsibility, which may generate a fork. It can then be later published and regularly maintained.

LSVerifier identified issues in Wireshark (OLIVEIRA, 2022i) related to array access (bounds), invalid pointer, and null pointer, which are associated to CWE-125, CWE-824, and CWE-476. All these issues were identified in the libraries CMake and network programming language (NPL), which are project dependencies, where dereference failures occurred due to out-of-bounds access and NP occurrences. The third-party library NPL is an ongoing project that has not been prioritized recently. The last significant update to this module was approximately nine years ago, with the latest reference in the commit logs dating back eight years. To maintain the robustness and security of Wireshark, the development team opted to remove this module.

We have not been able to validate the issues associated with VIM, Netdata, and OpenSSH. Indeed, some were classified as false positives (OLIVEIRA, 2022k; OLIVEIRA, 2022b; OLIVEIRA, 2022a; OLIVEIRA, 2022e) by developers, and others are still under discussion (OLIVEIRA, 2022c). However, we must mention that the related problems exist and suffer from the same problematic practices previously detailed in the OpenSSL

context.

Another issue involving multiple instances of invalid pointer dereference was reported for Putty. Numerous memory-related property violations, as detailed in Table 8, were identified. However, since the only communication channel for Putty is email, we have not received any feedback yet. Consequently, a given weakness may last long and cause much damage.

In our examination of TMUX, we filed an issue (OLIVEIRA, 2023a) addressing two distinct violations: null pointer dereference (CWE-476) and array out-of-bounds (CWE-125). The null pointer dereference was observed in the function, showcasing a critical vulnerability in the software, $cmd_refresh_client_update_subscription(...)$, where a null pointer returned by $strchr$ is dereferenced without a prior check. The array out-of-bounds issue was identified in the function $cmd_show_prompt_history_exec(...)$, where an index variable $type$ is used to access arrays without validating that it lies within the permissible range. Both issues can lead to undefined behavior and potential vulnerabilities within the application.

The analysis conducted for Redis revealed multiple violations (OLIVEIRA, 2023e; OLIVEIRA, 2023d). They included array bound violated (CWE-787), invalid pointer dereference (CWE-476), null pointer dereference (CWE-476), and access to object out-of-bounds (CWE-119). Some were confirmed as false positives (OLIVEIRA, 2023e). Although, for the current code structure, it is indeed true, other calls to the respective function may lead to the violation found here. Again, a simple check could avoid any future problems, even if they are intentional. Moreover, if an attacker tries it directly into the source code, he should also remove such a check, which would attract the attention of the respective code maintainers.

Anyway, a null pointer was identified in function $completionCallback(ls \rightarrow buf, \&lc)$ without confirming the non-nullity of ls and $ls \rightarrow buf$ (OLIVEIRA, 2023d). This oversight could lead to the dereferencing of a null pointer. It does pose a risk of undefined behavior if this function is called with a null pointer. Additionally, it should be noted that null checks may be adequate for ensuring pointer validity, as a pointer might seem legitimate but can be exploited to reference an invalid memory address. This

nuance highlights the importance of comprehensive pointer validation before usage. The developers dismissed this potential issue by claiming that a function or method will never be called in a certain way that triggers a problem despite confirmation of the issue. Consequently, this is a bad practice.

Some property violations were reported to the maintainers of SQLite (OLIVEIRA, 2023b; OLIVEIRA, 2023c). These issues are related to division by zero (CWE-369), array out-of-bound (CWE-787), same object violation (CWE-469), and null pointer dereference (CWE-476). The report from LSVerifier indicates a potential vulnerability in the internal function with signature $vdbePmaWriterInit(..., int\ nBuf, ...)$, in the file $vdbesort.c$, where a division by zero could occur if $nBuf$ is zero. The latter is used in a modulo operation, which, when its value is zero, results in an undefined behavior in C, leading to crashes or other unexpected behaviors. While developers might easily assert that the function $vdbePmaWriterInit()$ is never invoked with $nBuf \leq 0$, it's imperative to account for and mitigate such edge cases from a rigorous software engineering standpoint. This not only ensures code robustness but also preemptively addresses potential vulnerabilities.

It is worth mentioning that when we informed the respective developers that these results came from an automated analyzer, they started demonstrating disbelief. Indeed, the SQLite project's response to this violation reveals another prevalent behavior in the software development community. Developers often dismiss the results from static analyzers, labeling them as false positives. This perspective stems from the understanding that static analyzers frequently produce inaccurate results, leading to unnecessary alarms. The SQLite team's stance is clear: such reports will be disregarded without concrete evidence, such as an SQL script or specific code that can reproduce an issue. This approach, while pragmatic, is risky. Relying solely on tangible evidence might overlook potential vulnerabilities that haven't manifested yet but could be exploited. An over-reliance on a codebase's historical performance, as the SQLite team mentioned regarding their source tree's ability to confuse static analyzers, can lead to complacency.

It is essential to recognize that while static analyzers might produce false posi-

tives, they can also pinpoint genuine issues that might be overlooked during manual code reviews. Developers must strike a balance. While it's unreasonable to expect teams to act on every report from a static analyzer, completely ignoring them is not the solution. A more collaborative approach, where the reporter and the development team work together to validate and address potential issues, can lead to more secure and robust software. After all, the ultimate goal should be to ensure the software's integrity and safeguard it from potential threats, regardless of its origin.

Third-party libraries are the biggest problem, as seen in counterexample logs and bug report validations. Functions from other libraries that are called in software modules should always be carefully checked by developers before use, as they can be dangerous. Moreover, given that C programs often use pointers to access arrays, and those are usually passed as arguments to functions, such a condition can cause serious security issues.

Moreover, at this point, we have enough information to tackle our research goals again. As the respective software developers confirmed the problems found by LSVerifier, we can now completely answer **EG2**, confirming its feasibility for practical use.

The results show that LSVerifier is well-positioned in formal verification via BMC to check software vulnerabilities in large C-based software systems. This perception is also corroborated by the answers to our two research goals. Also, the prioritization algorithm enhances code analysis by function type, streamlining the identification of critical issues for efficient resolution. This verification tool is crucial because, despite developers' assurances that certain conditions are unlikely or impossible within the normal execution flow, our findings suggest otherwise. Confirmed vulnerabilities indicate that an attacker could feasibly trigger these improbable scenarios under certain manipulations, such as parameter tampering or code modification. Such an eventuality, previously dismissed by developers, could lead to significant and severe consequences if exploited. This reinforces the importance of our methodical approach, highlighting the need for rigorous security practices even in seemingly unlikely situations.

Indeed, LSVerifier successfully verified extensive software systems within a

reasonable time while avoiding high memory consumption. Nevertheless, the analysis performed here shows that further work is needed to reinforce and double-check vulnerabilities so one can have an upfront confirmation of what is presented. One alternative is to develop counterexample validators that are useful for developers of open-source applications. Such elements can help mitigate bad practices by showing that the refuted problems are not false positives. Without such validators, problem validation will depend solely on extensive analysis to confirm their existence.

Anyway, the LSVerifier's results may also be used as a mind-changing tool regarding software development practices. They show a high amount of existing vulnerabilities resulting from both careless coding and wrong behavior and practices, thus highlighting the need for diligent actions.

5.6 Summary

This section discusses the experimental evaluation of LSVerifier, a tool designed for verifying security vulnerabilities in large software systems written in C. The experiments were conducted on a personal computer with an Intel(R) Core(R) i7 CPU, running Ubuntu 20.04, to assess LSVerifier's performance and effectiveness in identifying common vulnerabilities within open-source software projects.

The experimental setup involved a comprehensive evaluation of LSVerifier using twelve prominent open-source programs, including VLC, VIM, Tmux, RUFUS, OpenSSH, CMake, Netdata, Wireshark, OpenSSL, PuTTY, SQLite, and Redis. These programs were selected based on their extensive code size, importance to the open-source community, and the use of linked third-party libraries. The experiments focused on individual analysis of all C files within these projects, checking each function for vulnerabilities using LSVerifier with specific flags for log plotting and ESBMC configuration.

The experimental evaluation of LSVerifier was driven by two primary objectives aimed at understanding its scalability and practical utility in the context of verifying security vulnerabilities in large software systems. The first goal, EG1, focused on evaluating the tool's ability to automate the verification process for extensive codebases

without the need for manual oversight, aiming to produce accurate results in a timely manner. This aspect was crucial in determining LSVerifier's efficiency and its potential to scale across different sizes of software projects. The second objective, EG2, sought to assess the tool's effectiveness in identifying common vulnerabilities that are recognized and acknowledged by the developers of the tested software modules. This goal aimed to gauge the practical applicability of LSVerifier in real-world scenarios, measuring its success in pinpointing issues that align with developers' experiences and insights, thereby validating its utility and effectiveness in enhancing software security.

The experimental validation of LSVerifier's effectiveness in identifying security vulnerabilities within large software systems faces potential challenges that could affect the study's validity, divided into three critical areas. First, the selection of benchmarks, involving open-source software projects, is pivotal to the evaluation process. Although the chosen projects represent complex systems widely utilized in the open-source realm, the results may not be universally applicable across all types of software, highlighting the need for a diverse range of benchmarks to comprehensively assess a verification tool's capability. Secondly, the performance and correctness of LSVerifier are under scrutiny, as the premise that function-by-function evaluation yields precise verification may not hold true in scenarios involving complex concurrency or parallelism. This could impact LSVerifier's efficiency and the accuracy of its verification process, especially considering the computational intensity required to examine all possible interactions between functions. Lastly, the process of counterexample validation presents its own set of difficulties. The complexity inherent in large software projects makes validating the identified vulnerabilities a formidable task, requiring meticulous testing and analysis to confirm the accuracy and reliability of the verification outcomes. Together, these factors outline the significant challenges in ensuring the experiments' validity and underscore the importance of addressing these threats to solidify the findings' credibility.

The application of LSVerifier to twelve prominent open-source C projects unearthed a wide array of vulnerabilities, with 'dereference failure' being the most common issue identified. The tool's ability to detect vulnerabilities, such as pointer dereferences, division by zero, dynamic object, and array-bound violations, attests to its

alignment with CWE standards and underscores its potential in enhancing software security. The experiments not only showcased LSVerifier's scalability in handling large codebases but also its efficiency in doing so within reasonable timeframes and without excessive memory consumption.

A deeper dive into the violated properties revealed through LSVerifier's analysis brought several vulnerabilities to light, some of which were acknowledged and rectified by the developers of the respective software projects. This process entailed the meticulous reporting of issues based on LSVerifier's counterexample traces and engaging with developers for validation. The constructive dialogue between LSVerifier's team and software maintainers emphasized the tool's practical impact in identifying and addressing real-world vulnerabilities. The commitment to resolving reported issues reflects the developers' recognition of LSVerifier's contributions to software security.

LSVerifier's experimental evaluation demonstrates its effectiveness in automating the verification process for large C-based software systems and its proficiency in identifying a comprehensive range of security vulnerabilities. The analysis of violated properties and the subsequent confirmation and rectification of issues underscore the tool's practical utility and the importance of rigorous security practices. Moving forward, enhancing LSVerifier with features such as counterexample validators could further bolster its efficacy by facilitating the validation of identified vulnerabilities, thus bridging the gap between automated verification and manual validation processes. The continued collaboration between verification tool developers and software maintainers will be crucial in advancing the state of software security in the open-source ecosystem.

6

CONCLUSIONS

Memory safety issues are a critical subset of software vulnerabilities that pose significant risks to software reliability and security. Memory safety in C programs pertains to the correct management of memory allocation, access, and deallocation, ensuring that programs only access memory allocated to them and preventing unauthorized access or modifications. The absence of inherent memory safety mechanisms in the C programming language often leads to vulnerabilities such as buffer overflows, use-after-free errors, and memory leaks. These vulnerabilities can be exploited by attackers to execute arbitrary code, lead to system crashes, or leak sensitive information. Addressing memory safety issues is essential for mitigating a wide range of software vulnerabilities in C, necessitating rigorous verification and validation techniques to identify and rectify such flaws before they can be exploited. The delicate balance between the performance benefits provided by direct memory management in C and the potential security risks highlights the ongoing challenge of ensuring memory safety while maintaining the efficiency and flexibility that C offers to developers.

In this research, we introduced LSVerifier, an innovative approach leveraging bounded model checking to identify security vulnerabilities in C open-source software projects. This work delineated its core functionalities, provided an in-depth evaluation of its architecture, and presented empirical results derived from real-world software applications. LSVerifier stands out by analyzing functions within C files and pinpointing critical errors such as improper pointer usage and memory access issues. Considering

the escalating security risks associated with third-party libraries in software development, this approach is pivotal.

LSVerifier's architecture is specifically tailored for large-scale systems, capable of handling extensive codebases like Wireshark, VLC, and CMake. Its methodology involves transforming code and properties into boolean formulas, which are then processed by a solver. If a property is violated, LSVerifier generates a counterexample, providing crucial insights into potential security breaches. The tool's design focuses on ease of use and efficiency. It scans project directories and performs individual function analyses, streamlining the verification process. The generated reports summarize the software weaknesses, offering a clear view of the violated properties and detailed logs from the analysis. This feature is particularly beneficial in human-in-the-loop verification methodologies.

We evaluated our tool and its associated algorithms using a dataset of ten large practical open-source C projects. In terms of software evaluation, we were able to find issues in databases with different sizes and target applications, ranging from tens of thousands to millions of code lines, which were confirmed by their respective maintainers. Such achievements confirmed our initial research goals and provided evidence of the efficacy of our methodology.

Furthermore, detailed analyses of counterexample logs and validated issue reports emphasize the imperative for rigorous scrutiny of functions, particularly those associated with pointers and arrays. These functions often harbor critical limitations, potentially escalating into significant security threats. Most of these vulnerabilities can be traced back to memory management discrepancies in third-party libraries, accentuating the pressing need for developers to adopt preventive strategies. By embracing defensive coding practices, utilizing memory-safe libraries, and ensuring rigorous boundary checks during memory operations, developers can prioritize secure memory management, substantially reducing the risk of security breaches in their software.

6.1 Future Works

In upcoming enhancements for LSVerifier, the focus will be on incorporating advanced technological approaches. The integration of automated parameter selection mechanisms and the application of machine learning techniques, especially Large Language Models (LLMs), are poised to refine the output analysis process significantly. Additionally, the introduction of functionalities for interrupting and resuming lengthy verification processes will add flexibility and efficiency. The utilization of cluster resources will also be a key aspect, aimed at accelerating the verification tasks. These improvements are geared towards making LSVerifier a more robust and efficient tool in the realm of software verification.

The evolution of LSVerifier is set to make substantial contributions to the field of software security, especially in open-source projects. Its enhanced ability to detect and report vulnerabilities will be pivotal in maintaining the integrity and reliability of software systems. As the software development landscape continues to evolve, the role of tools like LSVerifier becomes increasingly critical. These advancements not only signify progress in software verification techniques but also underscore the importance of continuous innovation in the face of growing and complex security challenges in software development.

BIBLIOGRAPHY

ALHAWI, O. M.; MUSTAFA, M. A.; CORDEIRO, L. C. Finding security vulnerabilities in unmanned aerial vehicles using software verification. In: *2019 International Workshop on Secure Internet of Things, SIoT 2019, Luxembourg, Luxembourg, September 26, 2019*. IEEE, 2019. p. 1–9. Disponível em: <https://doi.org/10.1109/SIOT48044.2019.9637109>. 20

ALJAAFARI, F. K. et al. Combining bmc and fuzzing techniques for finding software vulnerabilities in concurrent programs. *IEEE Access*, v. 10, p. 121365–121384, 2022. 38

ALSHMRANY, K. M. et al. Fusebmc: An energy-efficient test generator for finding security vulnerabilities in c programs. In: SPRINGER. *International Conference on Tests and Proofs*. [S.l.], 2021. p. 85–105. 15, 24, 38

ALSHMRANY, K. M. et al. Position paper: Towards a hybrid approach to protect against memory safety vulnerabilities. In: *IEEE Secure Development Conference, SecDev 2022, Atlanta, GA, USA, October 18-20, 2022*. IEEE, 2022. p. 52–58. Disponível em: <https://doi.org/10.1109/SecDev53368.2022.00020>. 14, 15

BAGUELIN, F. et al. *The OpenSSL punycode vulnerability (CVE-2022-3602): Overview, detection, exploitation, and remediation*. 2022. Disponível em: <https://securitylabs.datadoghq.com/articles/openssl-november-1-vulnerabilities/>. 68

BALDONI, R. et al. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 51, n. 3, p. 1–39, 2018. 36

BARANOVÁ, Z. et al. Model checking of c and c++ with divine 4. In: SPRINGER. *International Symposium on Automated Technology for Verification and Analysis*. [S.l.], 2017. p. 201–207. 35, 39

BARNEY, B. Posix threads programming. *National Laboratory*, v. 5, p. 46, 2009. 49

Barreto, R.; Cordeiro, L.; Fischer, B. Verifying embedded c software with timing constraints using an untimed bounded model checker. In: *2011 Brazilian Symposium on Computing System Engineering*. [S.l.: s.n.], 2011. p. 46–52. ISSN 2324-7894. 39

BARRETT, C. et al. Cvc4. In: SPRINGER. *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. [S.l.], 2011. p. 171–177. 23

BEYER, D. Advances in Automatic Software Verification: SV-COMP 2020. In: *Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.: s.n.], 2020. p. 347–367. ISBN 978-3-030-45237-7. 51

BEYER, D. Progress on software verification: Sv-comp 2022. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2022. p. 375–402. 17, 51

BEYER, D. Competition on software verification and witness validation: Sv-comp 2023. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2023. p. 495–522. 27

BEYER, D.; KEREMOGLU, M. E. Cpachecker: A tool for configurable software verification. In: SPRINGER. *International Conference on Computer Aided Verification*. [S.l.], 2011. p. 184–190. 35

BÖHME, M. et al. Directed Greybox Fuzzing. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. [S.l.: s.n.], 2017. p. 2329–2344. 15, 37

BOUDJEMA, E. H. et al. Detection of security vulnerabilities in c language applications. *Security and Privacy*, v. 1, n. 1, p. e8, 2018. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spy2.8>. 39

BRAUSSE, F. et al. ESBMC-CHERI: towards verification of C programs for CHERI platforms with ESBMC. In: RYU, S.; SMARAGDAKIS, Y. (Ed.). *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. ACM, 2022. p. 773–776. Disponível em: <https://doi.org/10.1145/3533767.3543289>. 39

BRUMMAYER, R.; BIERE, A. Boolector: An efficient smt solver for bit-vectors and arrays. In: SPRINGER. *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*. [S.l.], 2009. p. 174–177. 23

BÜNING, M. K.; SINZ, C.; FARAGÓ, D. Qpr verify: a static analysis tool for embedded software based on bounded model checking. In: SPRINGER. *Software Verification: 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20–21, 2020, Revised Selected Papers 13*. [S.l.], 2020. p. 21–32. 35

CADAR, C.; DUNBAR, D.; ENGLER, D. R. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*. [S.l.: s.n.], 2008. p. 209–224. 15

CAVADA, R. et al. The nuxmv symbolic model checker. In: SPRINGER. *International Conference on Computer Aided Verification*. [S.l.], 2014. p. 334–342. 35

CHENOY, A. et al. C-smc: A hybrid statistical model checking and concrete runtime engine for analyzing c programs. In: *SPIN 2021-27th International SPIN Symposium on Model Checking of Software*. [S.l.: s.n.], 2021. 35, 64

Cho, C. Y.; D'Silva, V.; Song, D. Blitz: Compositional bounded model checking for real-world programs. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2013. p. 136–146. 39

CHOI, Y. Safety analysis of trampoline os using model checking: an experience report. In: IEEE. *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. [S.l.], 2011. p. 200–209. 40

CLARKE, E.; KROENING, D.; LERDA, F. A Tool for Checking ANSI-C Programs. In: *Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.: s.n.], 2004. p. 168–176. 15, 37, 41

CLARKE, E.; KROENING, D.; LERDA, F. A tool for checking ansi-c programs. *Lecture Notes in Computer Science*, v. 2988, p. 168–176, 01 2004. 23

CMAKE. *CMake project*. 2000. Disponível em: <https://github.com/Kitware/CMake>. 60

COOK, B. et al. Using Model Checking Tools to Triage the Severity of Security Bugs in the Xen Hypervisor. In: *Formal Methods in Computer-Aided Design*. [S.l.: s.n.], 2020. 41

CORDEIRO, L.; FISCHER, B. Verifying multi-threaded software using smt-based context-bounded model checking. *ACM/IEEE 33rd International Conference on Software Engineering (ICSE)*, 2011. 15, 36

CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, IEEE, v. 38, n. 4, p. 957–974, 2011. 14, 15

Cordeiro, L.; Fischer, B.; Marques-Silva, J. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, v. 38, n. 4, p. 957–974, July 2012. ISSN 1939-3520. 37, 39

CORDEIRO, L. C.; FILHO, E. B. de L. Smt-based context-bounded model checking for embedded systems: Challenges and future trends. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 41, n. 3, p. 1–6, 2016. 14, 25

CORDEIRO, L. C.; FILHO, E. B. de L.; BESSA, I. V. de. Survey on automated symbolic verification and its application for synthesising cyber-physical systems. *IET Cyper-Phys. Syst.: Theory & Appl.*, v. 5, n. 1, p. 1–24, 2020. 14, 22

CORDEIRO, L. C. et al. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In: *International Conference on Embedded Software and Systems, ICESS*. [S.l.: s.n.], 2009. p. 396–403. 15

CORPORATION, T. M. *Common Weakness Enumeration (CWE)*. 2019. Accessed: 2023-08-20. Disponível em: <https://cwe.mitre.org/data/definitions/658.html>. 27

CORPORATION, T. M. *Common Weakness Enumeration (CWE)*. 2023. Accessed: 2023-07-10. Disponível em: <https://cwe.mitre.org/data/definitions/658.html>. 27, 28, 29, 30

COUSOT, P. Formal verification by abstract interpretation. In: SPRINGER. *NASA Formal Methods: 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings 4*. [S.l.], 2012. p. 3–7. 24

DEBUGGING, P.; PROFILING. *tracemalloc - Trace memory allocations*. 2023. Disponível em: <https://docs.python.org/3/library/tracemalloc.html>. 59

DEMPSEY, K. et al. *Automation Support for Security Control Assessments: Software Vulnerability Management*. National Institute of Standards and Technology, 2020. Disponível em: <https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8011-4.pdf>. 27

DINESH, S. et al. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In: IEEE. *2020 IEEE Symposium on Security and Privacy (SP)*. [S.l.], 2020. p. 1497–1511. 36

DUCKLIN, P. *Log4Shell explained – how it works, why you need to know, and how to fix it*. 2021. <https://nakedsecurity.sophos.com/2021/12/13/log4shell-explained-how-it-works-why-you-need-to-know-and-how-to-fix-it/>. 15

DUTERTRE, B.; MOURA, L. D. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, v. 2, n. 2, p. 1–2, 2006. 23

ECKERT, M. et al. {HeapHopper}: Bringing bounded model checking to heap implementation security. In: *27th USENIX Security Symposium (USENIX Security 18)*. [S.l.: s.n.], 2018. p. 99–116. 35, 40

FIORALDI, A.; D'ELIA, D. C.; QUERZONI, L. Fuzzing binaries for memory safety errors with qasan. In: IEEE. *2020 IEEE Secure Development (SecDev)*. [S.l.], 2020. p. 23–30. 35, 37

FISCHER, B. et al. Cbmc-ssm: Bounded model checking of c programs with symbolic shadow memory. In: *37th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2022. p. 1–5. 35

FOUNDATION, P. S. *glob — Unix style pathname pattern expansion*. 2021. Disponível em: <https://docs.python.org/3/library/glob.html>. 54

FOUNTOULAKIS, M. *netdata crash during queries #9713*. 2020. Disponível em: <https://github.com/netdata/netdata/issues/9713>. 68

GADELHA, M. et al. ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference. In: *Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.: s.n.], 2019. ISBN 978-3-030-17502-3. 15, 23

GADELHA, M. R.; CORDEIRO, L. C.; NICOLE, D. A. An efficient floating-point bit-blasting API for verifying C programs. In: CHRISTAKIS, M. et al. (Ed.). *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers*. Springer, 2020. (Lecture Notes in Computer Science, v. 12549), p. 178–195. Disponível em: <https://doi.org/10.1007/978-3-030-63618-0_11>. 14

GADELHA, M. R.; MENEZES, R. S.; CORDEIRO, L. C. Esbmc 6.1: automated test case generation using bounded model checking. *International Journal on Software Tools for Technology Transfer*, Springer, v. 23, n. 6, p. 857–861, 2021. 14, 25, 27, 35

GADELHA, M. Y. R. et al. Towards counterexample-guided k-induction for fast bug detection. In: LEAVENS, G. T.; GARCIA, A.; PASAREANU, C. S. (Ed.). *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018. p. 765–769. Disponível em: <https://doi.org/10.1145/3236024.3264840>. 32

GADELHA, M. Y. R. et al. Smt-based refutation of spurious bug reports in the clang static analyzer. In: ATLEE, J. M.; BULTAN, T.; WHITTLE, J. (Ed.). *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019. p. 11–14. Disponível em: <https://doi.org/10.1109/ICSE-Companion.2019.00026>. 22

GERHOLD, M.; HARTMANNS, A. Reproduction report for sv-comp 2023. *arXiv preprint arXiv:2303.06477*, 2023. 35

GERKING, C.; SCHUBERT, D.; BODDEN, E. Model checking the information flow security of real-time systems. In: SPRINGER. *International Symposium on Engineering Secure Software and Systems*. [S.l.], 2018. p. 27–43. 35, 39

GODEFROID, P. Fuzzing: hack, art, and science. *Commun. ACM*, v. 63, n. 2, p. 70–76, 2020. 15

GOSEVA-POPSTOJANOVA, K.; PERHINSCHI, A. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, Elsevier, v. 68, p. 18–33, 2015. 22

GREENBERG, A. Hackers remotely kill a jeep on the highway. *Wired*, July 2015. Disponível em: <https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/>. 14

GRITTI, F. et al. Heapster: Analyzing the security of dynamic allocators for monolithic firmware images. In: IEEE. *2022 IEEE Symposium on Security and Privacy (SP)*. [S.l.], 2022. p. 1082–1099. 35

GUEYE, A. et al. A decade of reoccurring software weaknesses. *IEEE Security & Privacy*, IEEE, v. 19, n. 6, p. 74–82, 2021. 21

GUI, B.; SONG, W.; HUANG, J. Uafsan: an object-identifier-based dynamic approach for detecting use-after-free vulnerabilities. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. [S.l.: s.n.], 2021. p. 309–321. 35, 37

HE, F.; SUN, Z.; FAN, H. Deagle: An smt-based verifier for multi-threaded programs (competition contribution). In: SPRINGER. *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II*. [S.l.], 2022. p. 424–428. 35

HOEPMAN, J.-H.; JACOBS, B. Increased security through open source. *Communications of the ACM*, ACM New York, NY, USA, v. 50, n. 1, p. 79–83, 2007. 15

HOLZMANN, G. J. Cobra: a light-weight tool for static and dynamic program analysis. *Innovations in Systems and Software Engineering*, Springer, v. 13, n. 1, p. 35–49, 2017. 41

INVERSO, O. et al. Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In: IEEE. *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2015. p. 807–812. 35

ISMAIL, H. I. et al. Dsverifier: A bounded model checking tool for digital systems. In: SPRINGER. *Model Checking Software: 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings 22*. [S.l.], 2015. p. 126–131. 35

IVANCIC, F. et al. Model Checking C Programs Using F-SOFT. In: . [S.l.: s.n.], 2005. v. 2005, p. 297 – 308. ISBN 0-7695-2451-6. 23

Kernel development community. *The frame buffer device*. [S.l.]: Linux, 2023. <https://docs.kernel.org/fb/framebuffer.html>. Accessed 15 August 2022. 73

KERNIGHAN, B.; RITCHIE, D. *The C Programming Language*. [S.l.]: Pearson, 2006. 14

KNORR, G.; DANZER, M.; UYTTERHOEVEN, G. *fbdev: video driver for framebuffer device*. [S.l.]: Linux, 2023. <https://linux.die.net/man/4/fbdev>. Accessed 15 August 2022. 73

KROENING, D.; TAUTSCHNIG, M. Cbmc–c bounded model checker. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2014. p. 389–391. 35

LANGE, T. et al. Ic3 software model checking. *International Journal on Software Tools for Technology Transfer*, Springer, v. 22, n. 2, p. 135–161, 2020. 35, 64

LEÓN, H. Ponce-de et al. Dartagnan: Bounded model checking for weak memory models (competition contribution). In: SPRINGER. *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II 26*. [S.l.], 2020. p. 378–382. 35

LHOMME, S. *fb: remove support*. Gitlab, 2022. Disponível em: <https://code.videolan.org/videolan/vlc/-/pipelines/227531>. 69, 71, 73

LOMUSCIO, A.; QU, H.; RAIMONDI, F. Mcmas: an open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer*, Springer, v. 19, n. 1, p. 9–30, 2017. 35

MALÍK, V. et al. 2ls: Arrays and loop unwinding: (competition contribution). In: SPRINGER. *Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22–27, 2023, Proceedings, Part II*. [S.l.], 2023. p. 529–534. 35

MANN, M. et al. Pono: a flexible and extensible smt-based model checker. In: SPRINGER. *International Conference on Computer Aided Verification*. [S.l.], 2021. p. 461–474. 35, 64

MATULEVICIUS, N.; CORDEIRO, L. C. Verifying security vulnerabilities for blockchain-based smart contracts. In: *XI Brazilian Symposium on Computing Systems Engineering, SBESC 2021, Florianopolis, Brazil, November 22-26, 2021*. IEEE, 2021. p. 1–8. Disponível em: <https://doi.org/10.1109/SBESC53686.2021.9628229>. 20

MEMARIAN, K. et al. Exploring c semantics and pointer provenance. *Proceedings of the ACM on Programming Languages*, ACM New York, NY, USA, v. 3, n. POPL, p. 1–32, 2019. 16

MENEZES, R. et al. ESBMC v7.4: Harnessing the power of intervals. *CoRR*, abs/2312.14746, 2023. Disponível em: <https://doi.org/10.48550/arXiv.2312.14746>. 25

MENEZES, R. et al. Map2check using LLVM and KLEE - (competition contribution). In: BEYER, D.; HUISMAN, M. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*. Springer, 2018. (Lecture Notes in Computer Science, v. 10806), p. 437–441. Disponível em: <https://doi.org/10.1007/978-3-319-89963-3_28>. 37

MERZ, F.; FALKE, S.; SINZ, C. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In: *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*. [S.l.: s.n.], 2012. (VSTTE'12), p. 146–161. ISBN 9783642277047. 23

MERZ, F.; FALKE, S.; SINZ, C. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In: SPRINGER. *International Conference on Verified Software: Tools, Theories, Experiments*. [S.l.], 2012. p. 146–161. 39

MITRE. *2023 CWE Top 25 Most Dangerous Software Weaknesses*. 2023. Accessed: 2023-07-10. Disponível em: <https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html>. 16, 51

MONTEIRO, F. R.; GADELHA, M. R.; CORDEIRO, L. C. Model checking c++ programs. *Software Testing, Verification and Reliability*, Wiley Online Library, v. 32, n. 1, p. e1793, 2022. 40

MORSE, J. et al. Context-bounded model checking of LTL properties for ANSI-C software. In: BARTHE, G.; PARDO, A.; SCHNEIDER, G. (Ed.). *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*. Springer, 2011. (Lecture Notes in Computer Science, v. 7041), p. 302–317. Disponível em: <https://doi.org/10.1007/978-3-642-24690-6_21>. 14

MOURA, L. D.; BJØRNER, N. Z3: An efficient smt solver. In: SPRINGER. *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2008. p. 337–340. 23

MUEGGE, S. M.; MURSHED, S. M. Time to discover and fix software vulnerabilities in open source software projects: Notes on measurement and data availability. In: IEEE. *2018 Portland International Conference on Management of Engineering and Technology (PICMET)*. [S.l.], 2018. p. 1–10. 20

NETDATA. *Netdata project*. 2006. Disponível em: <https://github.com/netdata/netdata>. 60

NGUYEN, T.-T. et al. Multiple program analysis techniques enable precise check for sei cert c coding standard. In: IEEE. *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. [S.l.], 2019. p. 70–77. 41

NIE, P.; JIANG, J.; MA, Z. Ctl symbolic model checking based on fuzzy logic. In: *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. [S.l.: s.n.], 2020. p. 380–385. 37

OHM, M. et al. Backstabber's knife collection: A review of open source software supply chain attacks. In: SPRINGER. *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. [S.l.], 2020. p. 23–43. 15

OLIVEIRA, J. *Bug 3382 - Software vulnerabilities detected using ESBMC-WR tool*. Bugzilla, 2022. Disponível em: <https://bugzilla.mindrot.org/show_bug.cgi?id=3382>. 69, 71, 74

OLIVEIRA, J. *Bug 3452 - Potential Software vulnerabilities detected using ESBMC-WR tool*. Bugzilla, 2022. Disponível em: <https://bugzilla.mindrot.org/show_bug.cgi?id=3452>. 69, 71, 74

OLIVEIRA, J. *[Bug]: Code properties violations found - dereference failure: invalid pointer #13219*. Github, 2022. Disponível em: <https://github.com/netdata/netdata/issues/13219>. 69, 71, 74

OLIVEIRA, J. *e2fsprogs issue*. [S.l.]: GitHub, 2022. <https://github.com/tytso/e2fsprogs/issues/103>. 69, 71, 72

OLIVEIRA, J. *Security Vulnerabilities found in sqlite3.c*. SQlite, 2022. Disponível em: <https://www.sqlite.org/forum/forumpost/3ffffb11d0>. 69, 71, 74

OLIVEIRA, J. *Software vulnerabilities detected during code analysis with ESBMC-WR #1856*. GitHub, 2022. Disponível em: <https://github.com/pbatard/rufus/issues/1856>. 69, 71

OLIVEIRA, J. *Software vulnerabilities detected during code analysis with ESBMC-WR tool*. Gitlab, 2022. Disponível em: <https://gitlab.kitware.com/cmake/cmake/-/issues/23132>. 71, 74

OLIVEIRA, J. *Software vulnerabilities detected during code analysis with ESBMC-WR tool #17560*. GitHub, 2022. Disponível em: <https://github.com/openssl/openssl/issues/17560>. 71, 73

OLIVEIRA, J. *Software vulnerabilities detected in development tools*. Gitlab, 2022. Disponível em: <https://gitlab.com/wireshark/wireshark/-/issues/17897>. 71, 74

OLIVEIRA, J. *Tiny-regex issue*. [S.l.]: GitHub, 2022. <https://github.com/kokke/tiny-regex-c/issues/76>. 69, 71

OLIVEIRA, J. *VIM issue*. Github, 2022. Disponível em: <https://github.com/vim/vim/issues/9571>. 71, 74

OLIVEIRA, J. *Code properties violations during software vulnerabilities investigation*. TMUX, 2023. Disponível em: <https://github.com/tmux/tmux/issues/3737>. 71, 75

OLIVEIRA, J. *Code properties violations during software vulnerabilities investigation - Bug report*. SQLite, 2023. Disponível em: <https://sqlite.org/forum/forumpost/ac645ab114>. 71, 76

OLIVEIRA, J. *Code properties violations during software vulnerabilities investigation - Bug report 2*. SQLite, 2023. Disponível em: <https://www.sqlite.org/forum/forumpost/a2d232d413>. 71, 76

OLIVEIRA, J. *Linenoise and lua issues*. Github, 2023. Disponível em: <https://github.com/janislley/lsverifier_final_results/blob/main/redis-7.0.11/issue%20report/Advisory_02.pdf>. 71, 75

OLIVEIRA, J. *Redis issue*. Github, 2023. Disponível em: <https://github.com/janislley/lsverifier_final_results/blob/main/redis-7.0.11/issue%20report/Advisory_01.pdf>. 71, 75

OORSCHOT, P. C. van. Memory errors and memory safety: C as a case study. *IEEE Security & Privacy*, IEEE, v. 21, n. 2, p. 70–76, 2023. 16

OPENSSH. *OpenSSH project*. 1999. Disponível em: <https://github.com/openssh/openssh-portable>. 44, 60

OPENSSL. *OpenSSL project*. 1998. Disponível em: <https://github.com/openssl/openssl>. 44, 60

PAGE, L. manual. 2023. Disponível em: <https://man7.org/linux/man-pages/index.html>. 59

PALMSKOG, K.; CELIK, A.; GLIGORIC, M. picoq: Parallel regression proving for large-scale verification projects. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. [S.l.: s.n.], 2018. p. 344–355. 21

PEREIRA, J. D.; VIEIRA, M. On the use of open-source c/c++ static analysis tools in large projects. In: *16th European Dependable Computing Conference (EDCC)*. [S.l.: s.n.], 2020. p. 97–102. 15

PHILIPPAERTS, P. et al. Software verification with verifast: Industrial case studies. *Science of Computer Programming*, Elsevier, v. 82, p. 77–97, 2014. 35

PIRA, E.; RAFE, V.; NIKANJAM, A. Emcdm: Efficient model checking by data mining for verification of complex software systems specified through architectural styles. *Applied Soft Computing*, Elsevier, v. 49, p. 1185–1201, 2016. 35

PLATE, H.; PONTA, S. E.; SABETTA, A. Impact assessment for vulnerabilities in open-source software libraries. In: IEEE. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2015. p. 411–420. 20, 21

PONTA, S. E.; PLATE, H.; SABETTA, A. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, Springer, v. 25, n. 5, p. 3175–3215, 2020. 21

PUTTY. *Xen project*. 1999. Disponível em: <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>. 44, 54, 60

RAKAMARIĆ, Z.; EMMI, M. Smack: Decoupling source language details from verifier implementations. In: SPRINGER. *International Conference on Computer Aided Verification*. [S.l.], 2014. p. 106–113. 35

REDIS. *Redis project*. 2009. Disponível em: <https://github.com/redis/>. 60

RICHARDSON, A. *Complete Spatial Safety for C and C++ using CHERI Capabilities*. [S.l.], 2020. 38, 39

RICHTER, C.; WEHRHEIM, H. Pesco: Predicting sequential combinations of verifiers. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2019. p. 229–233. 35

RIVAL, X.; YI, K. *Introduction to static analysis: an abstract interpretation perspective*. [S.l.]: Mit Press, 2020. 36

ROCHA, H. et al. Understanding programming bugs in ansi-c software using bounded model checking counter-examples. In: *Integrated Formal Methods*. [S.l.: s.n.], 2012. p. 128–142. ISBN 978-3-642-30729-4. 39

ROCHA, H. et al. Map2Check: Using Symbolic Execution and Fuzzing. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.: s.n.], 2020. p. 403–407. 36, 37

ROČKAI, P. et al. Divm: model checking with llvm and graph memory. *Journal of Systems and Software*, Elsevier, v. 143, p. 1–13, 2018. 35

RODRIGUEZ, M.; PIATTINI, M.; EBERT, C. Software verification and validation technologies and tools. *IEEE Software*, IEEE, v. 36, n. 2, p. 13–24, 2019. 14

RUFUS. *RUFUS project*. 2011. Disponível em: <https://github.com/pbatard/rufus>. 60

RUSCIO, D. D.; PELLICCIONE, P.; PIERANTONIO, A. Evoss: A tool for managing the evolution of free and open source software systems. In: IEEE. *2012 34th International Conference on Software Engineering (ICSE)*. [S.l.], 2012. p. 1415–1418. 21

SECURITY, S. *NULL Pointer Dereference*. 2022. Disponível em: <https://security.snyk.io/vuln/SNYK-CENTOS6-SQLITE-3010350>. 68

SITU, L. et al. Vanguard: Detecting missing checks for prognosing potential vulnerabilities. In: *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*. [S.l.: s.n.], 2018. p. 1–10. 35, 37

SOUSA, J. O. de et al. Lsverifier: A bmc approach to identify security vulnerabilities in c open-source software projects. In: SBC. *Anais Estendidos do XXIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*. [S.l.], 2023. p. 17–24. 35

SQLITE. *SQLite project*. 2000. Disponível em: <https://github.com/sqlite/sqlite>. 60

STANDARDIZATION, I. O. for. Iso/iec 9899-2011: Programming languages – c. *ISO Working Group, Geneva, Switzerland*, 2012. 51

SUCKEVIC, G. de A. *Bug 1841231 (CVE-2020-13435) - CVE-2020-13435 sqlite: NULL pointer dereference in sqlite3ExprCodeTarget()*. 2023. Disponível em: <https://bugzilla.redhat.com/show_bug.cgi?id=1841231>. 68

TAN, L. et al. Bug characteristics in open source software. *Empirical software engineering*, Springer, v. 19, p. 1665–1705, 2014. 15

TIHANYI, N. et al. The formai dataset: Generative AI in software security through the lens of formal verification. In: MCINTOSH, S.; CHOI, E.; HERBOLD, S. (Ed.). *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2023, San Francisco, CA, USA, 8 December 2023*. ACM, 2023. p. 33–43. Disponível em: <https://doi.org/10.1145/3617555.3617874>. 14

TMUX. *TMUX project*. 2007. Disponível em: <https://github.com/tmux/tmux>. 60

VANROSSUM, G. *Python reference manual*. [S.l.]: Department of Computer Science [CS], 1995. 51

VEEN, V. van der et al. Memory Errors: The Past, The Present, and The Future. In: *Research in Attacks, Intrusions, and Defenses*. [S.l.: s.n.], 2012. p. 86–106. ISBN 978-3-642-33338-5. 15

VIM. *VIM project*. 1991. Disponível em: <https://github.com/vim/vim>. 60

VLC. *VLC project*. 2001. Disponível em: <https://github.com/videolan/vlc>. 60

VOROBYOV, K.; KOSMATOV, N.; SIGNOLES, J. Detection of security vulnerabilities in c code using runtime verification: An experience report. In: *Tests and Proofs*. [S.l.: s.n.], 2018. p. 139–156. ISBN 978-3-319-92994-1. 14, 37

WEN, S.-F. Software security in open source development: A systematic literature review. In: IEEE. *2017 21st conference of open innovations association (fruct)*. [S.l.], 2017. p. 364–373. 20

WIRESHARK. *Wireshark project*. 1998. Disponível em: <https://gitlab.com/wireshark/wireshark>. 60

XIAO, S.; WITSCHEY, J.; MURPHY-HILL, E. Social influences on secure development tool adoption: why security tools spread. In: *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*. [S.l.: s.n.], 2014. p. 1095–1106. 21

YAMAGUCHI, F. et al. Chucky: Exposing missing checks in source code for vulnerability discovery. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. [S.l.: s.n.], 2013. p. 499–510. 35

YAN, H. et al. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In: IEEE. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. [S.l.], 2018. p. 327–337. 35

ZOU, J. et al. Research on secure stereoscopic self-checking scheme for open source software. In: *Proceedings of the 2019 International Conference on Artificial Intelligence and Computer Science*. [S.l.: s.n.], 2019. p. 158–162. 21