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

Janislley Oliveira de Sousa[1,2], Bruno Carvalho de Farias[3], Thales Araujo da Silva[2], Eddie Batista de Lima Filho[2,4], Lucas Carvalho Cordeiro[2,3]

[1]Sidia Institute of Science and Technology, Manaus, Brazil

[2]Federal University of Amazonas (UFAM), Manaus, Brazil

[3]University of Manchester, Manchester, United Kingdom

[4]TPV Technology, Manaus, Brazil

janislley.sousa@sidia.com, bruno.farias@manchester.ac.uk,

thales.tas@gmail.com, eddie.filho@tpv-tech.com,

lucascordeiro@ufam.edu.br

Abstract. *Researchers continue to advance the state-of-the-art of software vulnerability analysis. Software validation and verification techniques are indispensable tools for cultivating robust systems characterized by high levels of dependability and reliability. Remarkably, the pressing concern of memory errors in C software looms large in the landscape of systems security. This paper introduces the innovative tool called LSVerifier, leveraging the bounded model checking technique to effectively detect security vulnerabilities in C open-source software. The proposed tool emerges as a pivotal asset for identifying vulnerabilities and generating an output report summarizing the software weaknesses found. The method's efficacy was validated through real-world applications. The results show that LSVerifier was able to check complex open-source software, identifying software issues that could potentially result in vulnerabilities.*

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 et al. 2019]. The implications of software vulnerabilities are not merely confined to technical malfunctions but extend to potentially catastrophic consequences. Airbus discovered a software vulnerability in the A400M aircraft, leading to a crash in 2015. The fault, originating in the engine control units, caused the engines to power off shortly after take-off [Guardian 2015]. Also, security researchers could remotely exploit a vulnerability in the Jeep Cherokee's Uconnect infotainment system. By gaining access to the system, they could take over various vehicle functions, including engine and brakes [Wired 2015]. 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.

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 [van 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.

Here, we introduce LSVerifier, which is a tool designed to help end users better understand the security vulnerability issues in C open-source software projects, using Bounded Model Checking (BMC) and Satisfiability Modulo Theories (SMT) to identify security bugs. It verifies all source-code files in a specific open-source module or application. We implemented LSVerifier as a Python tool compatible with the efficient SMT-based context-bounded model checker (ESBMC), whose goal is to examine extensive open-source software. By analyzing files and functions individually and verifying them with ESBMC, it seeks to uncover vulnerabilities, providing an output report of its findings.

The remainder of this article is organized as follows. In Section 2, the essential background is provided. Section 3 outlines the tool's design, while Section 4 comprehensively demonstrates the tool's functionalities through experiments. Lastly, Section 5 presents our conclusions and achievements.

2. Background

In this section, we present the concepts and technologies, addressing the aspects of their fundamental structure and implementation essential for the LSVerifier tool.

2.1. 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 et al. 2004, Merz et al. 2012, Gadelha et al. 2019, Ivancic et al. 2005], and properties under verification are defined as follows:

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

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. The Underlying Checker

ESBMC [Gadelha et al. 2021], as employed in this study, is a robust and openly available tool that serves as our chosen BMC module for software verification. This mature model checker is designed for the verification of 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 and de Lima Filho 2016].

3. LSVerifier Tool

In the following sections, we will go through the architecture and key features of LSVerifier, highlighting foundational principles and concepts for the implementation approach.

3.1. Architecture and Main Functionalities

LSVerifier conducts a comprehensive verification process, as shown in Figure 1. It specifies the target source-code directory and the required LSVerifier configuration, including solver, encoding, and verification methods. Subsequently, all `.c` files are listed and examined using ESBMC, which leads to creating a spreadsheet summarizing the obtained results.

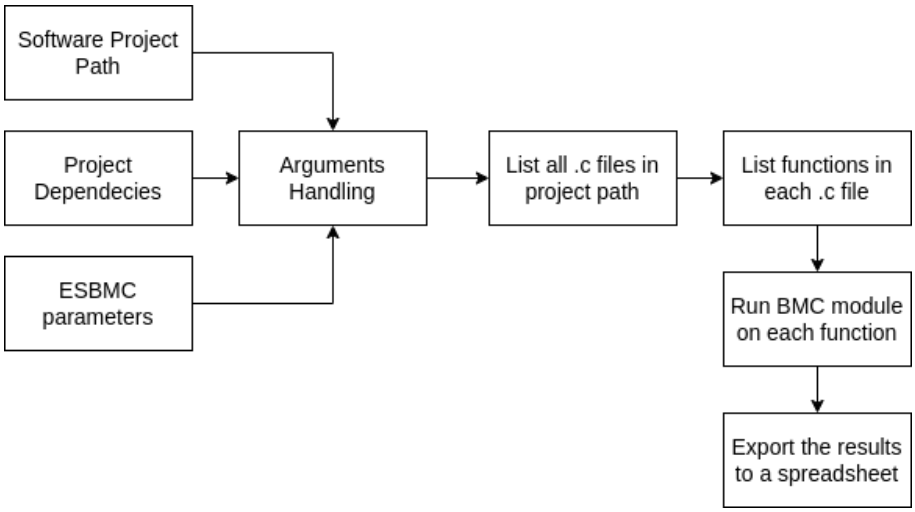

Figure 1. LSVerifier verification process.

The LSVerifier's configuration parameters are categorized into five groups: (1) file listing, (2) function verification, (3) outcome display, (4) ESBMC options, and (5) invalid pointer retest. These parameters provide the necessary information for LSVerifier to verify files and generate a final verification report. The available options are as follows:

- **'-e'** configures ESBMC regarding verification and options;
- **'-i'** provides a path for a file's specific dependencies (library dependencies) that is forwarded to ESBMC;
- **'-f'** enables verification in a function-by-function fashion;
- **'-v'** enables the verbose mode;
- **'-r'** enables the recursive mode, where all "$.c$" files in the existing sub-directories are listed;
- **'-d'** enables verification on a specific directory;
- **'-fl'** enables verification in just a single file;
- **'-h'** shows the available options.

3.2. Implementation Details

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

LSVerifier efficiently verifies software written in the C language, extending its reach to third-party libraries. It processes software source code P in directory D using configuration C. It starts by listing all `.c` files in D, doing this recursively based on the configuration provided earlier. Each file's declared functions F are then identified and subsequently verified by ESBMC, looking for various property violations (e.g., pointer safety, arithmetic overflow, and array bounds errors). Any violations found during this procedure are communicated to a control script. Upon verification completion, a detailed report is generated in a spreadsheet V.

LSVerifier aims to provide comprehensive support for the entire C11 standard [for Standardization 2012], which stands as the prevailing standard for the C programming language. This tool operates by identifying vulnerabilities in software through the simulation of a finite program execution prefix encompassing all conceivable defined inputs. The verification process also explicitly exploits interleavings, generating one symbolic execution per interleaving. By default, LSVerifier checks for pointer safety, array out-of-bounds, division by zero, and assertions a user specifies.

Some software modules may require specific dependencies to be manually listed before using LSVerifier. All dependency paths must be included in a file and provided using the parameter *-i*, with each dependency listed on a separate line, as illustrated in Figure 2.

```
1  /usr/include/glib-2.0/
2  /usr/lib/x86_64-linux-gnu/glib-2.0/include/
3  extcap/
4  plugins/epan/ethercat/
5  plugins/epan/falco_bridge/
6  plugins/epan/wimaxmacphy/
7  randpkt_core/
8  writecap/
9  epan/crypt/
10 ...
```

Figure 2. An example of the dependency file used for the software Wireshark.

When analyzing an entire project, parameter configuration to define library dependencies is required as an argument in the command line, as described below.

```
$ lsverifier -r -v -f -e "--unwind 1
--no-unwinding-assertions" -i dep.txt
```

Moreover, LSVerifier can verify specific "*.c*" files as illustrated below. With this command, it will check all functions, which are passed as an input argument, as described below.

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

In addition, the tool can verify specific properties, such as memory leaks, as described below.

```
$ lsverifier -v -f -fl main.c -e "1 --memory-leak-check"
```

Figure 3 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 included in a report file and saved in a directory "/output".

```
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 3. Example of counterexample created by LSVerifier.

4. Experiments

All experiments described in this work were conducted on an Intel(R) Core(R) i7 CPU 9750H operating at 2.60 GHz, with 32 GB of RAM, and running the Ubuntu OS. As benchmarking, we prepared a dataset consisting of five commonly used software modules based on the C language: RUFUS, OpenSSH, CMake, Wireshark, and PuTTY. All open-source software code used here was distributed under Open-source licenses (GNU GPL, Apache, and MIT). More details for each program can be found in each benchmark's repository[1].

The command below was used to run LSVerifier for the entire dataset's software to validate the proposed approach.

```
$ lsverifier -r -v -f -e "--unwind 1
--no-unwinding-assertions" -i dep.txt
```

[1] https://github.com/janislley/LSVerifier_Benchmarks

We were able to verify each function of each software module. The verification process resulted in several property violations. Table 1 shows all property violations found during our experiments. Most of them are related to the MITRE's "Top 25" CWE list [MITRE 2023](*i.e.*, pointer dereference, division by zero, dynamic object violation, and array bound violation). Also, the verification analysis consists of the number of vulnerabilities, the number of files, and functions verified according to memory consumption and analysis time. The same table compares five software applications: VIM, RUFUS, OpenSSH, Wireshark, and PuTTY, focusing on various metrics. PuTTY had the highest number of violated properties, i.e., 2019, indicating a possible difference in complexity or adherence to standards. Wireshark leads in the number of files and verified functions, i.e., 2194 and 108824, respectively. OpenSSH, VIM, and RUFUS show moderate numbers, while PuTTY's count is low compared to its high number of property violations.

Table 1. Dataset analysis using LSVerifier tool.

Software	Property violations	Files analyzed	Functions verified	Overall time	Peak memory usage
VIM	5	184	8804	406.02 s	36.46 MB
RUFUS	186	142	1575	101.59 s	32.6 MB
OpenSSH	337	286	3033	490.33 s	15.32 MB
Wireshark	122	2194	108824	39413.97 s	119.52 MB
PuTTY	2019	244	4575	91448.89 s	53.79 MB

LSVerifier maintained low peak-memory usage, and the overall verification time exceeded our expectations, considering the number of files analyzed in each software. The disparity between the overall time and the number of files/functions checked doesn't account for the inherent complexity of individual software code units. Some functions or files may involve intricate logic, nested structures, or dependencies, leading to a higher computational load during analysis. As a result, even if the number of files/functions is similar, the time required to thoroughly assess each component can vary significantly. Larger functions may require more time for analysis due to their complexity. Functions with intricate conditional statements, loops, or extensive variable interactions may increase computational time. The efficiency of CTAGS and the volume of information it processes can contribute to variations in time. For instance, certain code structures might lead to longer processing times due to the way CTAGS parses them.

Moreover, an investigation was conducted to determine the underlying causes of property violations within each software module. Some may constitute vulnerabilities and are often linked to instances where the source code semantics are undefined, such as invalid access to pointers or arrays. During our analysis, various property violations were discovered across different software. RUFUS presented property violations such as array out of bounds, with three issues opened and fixed[2] regarding imported libraries. The Wireshark's property violations, which are related to array out of bounds and invalid pointers, were due to errors in the NPL third-party library, and a fix[3] was provided to remove this dependency. These findings emphasize the recurring theme of vulnerabilities that come from third-party libraries. In addition, they clearly show the importance of careful inspection and prompt resolution of such issues.

Our experimental results show that LSVerifier was effective when evaluating pub-

[2]https://github.com/pbatard/rufus/issues/1856

[3]https://gitlab.com/wireshark/wireshark/-/issues/17897

licly available benchmarks. Moreover, it provided a complete report with counterexamples that can be used to reproduce the identified errors. One of our outstanding results is using counterexamples to support engineers in writing security tests to further analyze bugs.

5. Final Considerations

Demonstration

The source code, documentation, usage instructions, and installation information are available in the LSVerifier's repository[4]. Its demonstration will occur within a local environment running on a Linux machine. Its functionality will be showcased through the following steps: (a) presentation of the tool's execution parameters; (b) introduction of the directory structure employed; (c) demonstration of its verification capabilities to identify property violations in C code that could lead to a security vulnerability; (d) detailed showcasing and demonstration of the analysis report generated by it; and (e) presentation of results (i.e., tables, figures, verification reports).

Conclusion

In this work, we presented LSVerifier, a new approach using a bounded model checker to exploit security vulnerabilities in C open-source software projects. We described its functionalities, evaluated its implementation, and showed experiments performed with real software. In summary, third-party libraries are emerging as a main concern in software security, with many issues detected by LSVerifier. The findings from counterexample logs and bug report validations highlight the necessity for meticulous examination of functions used in software, especially those involving pointers and arrays, as they present severe limitations that may lead to serious security risks. Most of the vulnerabilities stem from memory management mishaps in third-party libraries, which shows the urgent need for developers to implement preventative measures. Through defensive programming practices, using memory-safe libraries, and the execution of boundary checks on memory access, developers can focus on secure memory management and substantially mitigate potential security vulnerabilities in their projects.

In terms of future work, there are several promising directions to explore. First, the integration of a prioritization approach within LSVerifier can significantly enhance its utility. This would involve developing algorithms to prioritize the checking of software vulnerabilities based on their potential impact, thereby enabling developers to focus on the most critical issues first. Second, we plan to implement a feature that allows users to check for a specific class of vulnerabilities, such as buffer overflows or pointer dereference failures. Ultimately, these improvements aim to make LSVerifier a more versatile and effective tool for identifying and mitigating security risks in software projects, particularly those associated with third-party libraries.

Acknowledgment

The authors are grateful for the support offered by the SIDIA R&D Institute in the Model project. This work was partially supported by Samsung, using resources of Informatics Law for Western Amazon (Federal Law No. 8.387/1991). Therefore, the present work disclosure is in accordance as foreseen in article No. 39 of number decree 10.521/2020.

[4]`https://github.com/janislley/LSVerifier`

References

[Clarke et al. 2004] Clarke, E., Kroening, D., and Lerda, F. (2004). A tool for checking ansi-c programs. Lecture Notes in Computer Science, 2988:168–176.

[Cordeiro and de Lima Filho 2016] Cordeiro, L. C. and de Lima Filho, E. B. (2016). Smt-based context-bounded model checking for embedded systems: Challenges and future trends. ACM SIGSOFT Software Engineering Notes, 41(3):1–6.

[for Standardization 2012] for Standardization, I. O. (2012). Iso/iec 9899-2011: Programming languages – c. ISO Working Group, Geneva, Switzerland.

[Gadelha et al. 2019] Gadelha, M., Monteiro, F., Cordeiro, L., and Nicole, D. (2019). ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference. In Tools and Algorithms for the Construction and Analysis of Systems.

[Gadelha et al. 2021] Gadelha, M. R., Menezes, R. S., and Cordeiro, L. C. (2021). Esbmc 6.1: automated test case generation using bounded model checking. International Journal on Software Tools for Technology Transfer, 23(6):857–861.

[Guardian 2015] Guardian, T. (2015). Airbus issues software bug alert after fatal plane crash. Available at: `https://www.theguardian.com/technology/2015/may/20/airbus-issues-alert-software-bug-fatal-plane-crash`. Accessed on 2023-07-25.

[Ivancic et al. 2005] Ivancic, F., Shlyakhter, I., Gupta, A., Ganai, M. K., Kahlon, V., Wang, C., and Yang, Z. (2005). Model Checking C Programs Using F-SOFT. volume 2005, pages 297 – 308.

[Merz et al. 2012] Merz, F., Falke, S., and Sinz, C. (2012). 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, VSTTE'12, page 146–161.

[MITRE 2023] MITRE (2023). 2023 cwe top 25 most dangerous software weaknesses. Accessed: 2023-07-10.

[Rodriguez et al. 2019] Rodriguez, M., Piattini, M., and Ebert, C. (2019). Software verification and validation technologies and tools. IEEE Software, 36(2):13–24.

[van Oorschot 2023] van Oorschot, P. C. (2023). Memory errors and memory safety: C as a case study. IEEE Security & Privacy, 21(2):70–76.

[Wired 2015] Wired (2015). Hackers remotely kill a jeep on the highway—with me in it. Available at: `https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/`. Accessed on 2023-07-25.