Memory Leak Detection Tool

Vaishali Jabade  
*Department of Electronics and Telecommunication*  
Vishwakarma Institute of TechnologyPune, India  
[vaishali.jabade@vit.edu](mailto:vaishali.jabade@vit.edu)

Sanket Khade  
*Department of Electronics and Telecommunication*  
Vishwakarma Institute of TechnologyPune, India  
[sanket.khade21@vit.edu](mailto:sanket.khade21@vit.edu)Parth Kharade   
*Department of Electronics and Telecommunication*  
Vishwakarma Institute of TechnologyPune, India  
[parth.kharade21@vit.edu](mailto:parth.kharade21@vit.edu)

Karan Walekar  
*Department of Electronics and Telecommunication*  
*Vishwakarma Institute of Technology*Pune, India  
[santosh.karan21@vit.edu](mailto:santosh.karan21@vit.edu)

*Abstract*— Memory leaks pose significant challenges in software development, often leading to degraded system performance and reliability issues. Traditional static analysis tools struggle to effectively detect memory leaks, especially in complex codebases with intricate control flows. This paper introduces a novel approach to address this challenge by proposing a projection-based method for detecting memory leaks in C source code. Leveraging the inherent features of memory allocation and deallocation in C, the proposed approach simplifies the original control flow graph, thereby reducing analysis complexity. Furthermore, a memory-leak detection tool, termed PML Checker, is implemented and evaluated against existing static analysis tools using public benchmarks and custom test cases. Experimental results demonstrate that PML Checker outperforms other tools in detecting memory leak vulnerabilities, particularly in scenarios involving complex control flows and data types, achieving higher efficiency and accuracy. This research contributes to enhancing software safety and reliability by offering a more effective approach to detect memory leaks in C codebases.

Keywords— Memory Tracking, Memory Leak Detection, Memory Allocation, Dynamic Memory Management, Resource Management.

# Introduction

In the realm of software development, memory leaks stand as persistent adversaries, particularly in codebases authored in C and C++. These elusive issues, characterized by their inconspicuous symptoms and elusive origins, pose formidable challenges to developers, often resulting in program crashes due to the gradual depletion of finite memory resources. This concern is especially acute in the domain of embedded systems, encompassing devices such as mobile phones, microcontrollers, and sensors, where optimal resource utilization is paramount.Memory leaks manifest in two primary forms: forgotten memory and lost memory. Forgotten memory, also known as dead memory, refers to memory that remains allocated but is no longer actively accessed by the program. Conversely, lost memory, or unreachable memory, denotes memory areas that have become inaccessible to the program, leading to potential resource waste. Detecting and mitigating these leaks has spurred significant research efforts, spanning both static and dynamic analysis methodologies.Static analysis techniques involve scrutinizing program code before execution, leveraging control flow graphs to identify potential memory leaks. While offering comprehensive coverage of program paths, static analysis suffers from computational overhead and inherent imprecision, often leading to false positives or negatives. Dynamic approaches, on the other hand, track memory allocation and deallocation during program execution, offering real-time insights into memory utilization. Despite their efficacy, existing dynamic tools typically pinpoint leak locations without providing actionable insights for remediation.Motivated by the limitations of existing methodologies, this paper introduces an innovative approach to detect, eliminate, and fix memory leaks through a fusion of dynamic analysis and dynamic symbolic execution (DSE). Our approach entails instrumenting the target program before execution, capturing relevant memory allocation data during runtime, and employing DSE to explore diverse program execution paths comprehensively.To assess the effectiveness of our memory leak detection and remediation solution, we developed a custom tool tailored specifically for our project. This tool, dubbed "Leak Finder," was meticulously crafted to provide comprehensive memory leak detection and remediation functionalities for C programs.In summary, this paper makes three significant contributions: proposing a novel approach for memory leak detection and resolution, implementing a corresponding tool for practical application, and conducting extensive experiments to validate the efficacy of the proposed methodology. The subsequent sections delve into the details of our approach, experimental setup, and comparative analysis with existing techniques, culminating in a comprehensive review of our findings and avenues for future research.

# Related works

[1] The paper proposes a novel approach to detect memory leaks in C source code, specifically targeting programs with complex control flows. The authors introduce a projection algorithm that simplifies the control flow graph of a program, reducing the analysis complexity and improving the efficiency and accuracy of memory leak detection. They implement a memory leak detection tool, PML Checker, and evaluate it against three open-source static detection tools on public benchmarks and study test cases. The results show that PML Checker outperforms the other tools in detecting memory leak vulnerabilities in complex control flows and complex data types, while also achieving higher efficiency and accuracy on public benchmarks.

[2] The paper proposes an ownership inference algorithm for C and C++ programs, focusing on identifying potential memory leaks and double deletes. It introduces constraints, ownership graphs, and an interprocedural analysis method to determine ownership types for methods and fields. The algorithm iteratively evaluates method ownership types, handles control flow, and simplifies constraints to create a canonical ownership graph for each method. It also addresses challenges like handling system calls and ordering constraints based on precision to minimize error propagation. The approach prioritizes more precise constraints and classifies constraints based on their reliability to improve the accuracy of identifying ownership errors.

[3] The paper discusses memory leak detection algorithms, focusing on linear regression-based approaches like Linear Backward Regression (LBR) and Linear Backward Regression based on Change Points Detection (LBRCPD). These algorithms analyze memory utilization data to detect patterns indicative of memory leaks. Additionally, the document introduces the Precog algorithm, which combines offline training and online detection phases to identify anomalous memory usage trends. The algorithms aim to automate the detection of memory leaks in deployed applications using system memory utilization data.

[4] The paper proposes a method for identifying and fixing memory leaks in C programs. The authors classify procedures into alloc, use, and dealloc procedures and develop an algorithm to build summaries of four sets for each procedure that records which nodes on the points-to graph are allocated, escaped, used, and deallocated. The algorithm is then used to identify the types of a procedure with respect to a particular allocation and insert deallocation statements to fix leaks. The paper also discusses the challenges of identifying procedures and global variables and presents a refinement to the points-to graphs to avoid useless deallocation statements. The leak detection and fix stage includes a forward dataflow analysis to find what deallocations may have been reached before or at each CFG node, a backward dataflow analysis to find what deallocations and uses of allocations may be reached after or at each CFG node, a traversal on the edges to identify the variables that only point to the allocations that can be deallocated at each CFG edge, and a forward, greedy algorithm that aims to select the earliest points to insert deallocations.

[5] The article proposes a dynamic approach for detecting, eliminating, and fixing memory leaks in programs using the MC compiler. The approach involves instrumenting the code to collect memory information during execution, which is then used to detect leaks, record leak locations, and eliminate and fix leaks. The instrumentation collects information about allocated addresses, allocation locations, allocated sizes, pointer values, and deallocation addresses. The back-end leak checker calculates the number of variables pointing to each allocated memory area for leak detection and records leak locations and changes to variables pointing to each leaked memory area for leak elimination and fixing. The approach is effective in detecting and eliminating memory leaks in programs.

[6] The paper provides a detailed examination of memory leak detection in Java, categorizing methods into online and offline approaches. Online methods actively monitor running applications to detect leaks but can be resource-intensive and require specific JVM modifications. Offline methods, like heap dump analysis, offer low overhead but lack real-time information. Visualization techniques aid in spotting memory issues. The comparison section evaluates methodologies based on metrics, performance, and intrusiveness. Various tools and techniques, such as object ownership profiling and growth analysis, are discussed, highlighting the importance of understanding memory usage patterns and identifying memory leaks efficiently.

[7] The paper discusses Saturn, a static leak detector based on path-sensitive pointer analysis. It outlines how Saturn analyzes function bodies, handles loops, and performs interprocedural analysis. The detector targets memory leaks, particularly neglecting to free allocated memory blocks. It uses function summaries to infer memory behavior and escaping objects. The analysis involves tracking access paths, defining escape and leak conditions, and generating function summaries. Experimental results show the detector's effectiveness in analyzing user space applications like Samba, OpenSSL, PostFix, Binutils, OpenSSH, and the Linux kernel, achieving scalability and high precision in leak detection.

[8] The document discusses memory leak detection using full-sparse value-flow analysis, focusing on the conversion to SSA form, a motivating example, pre-analysis using Andersen-style pointer analysis, building full-sparse SSA form, generating regions for a program, and leak detection through SVFG analysis. It introduces the concepts of µ and χ functions, region identification, context-sensitive reachability analysis, and sink reachability algorithms using BDDs. The process involves forward and backward slices in the SVFG, all-path analysis to detect leaks, and the computation of guards on edges to capture path conditions for value flow. Saber's approach utilizes BDDs for path encoding, and the document details the steps involved in solving the all-path graph reachability problem.

[9] The paper introduces a novel memory leak detection method that utilizes a symbol table to track program scope and memory behaviors, constructs a state machine to detect memory leaks, and employs algorithms for defect pattern matching and function summary generation to enhance detection accuracy.By establishing a state machine, utilizing defect pattern matching, and generating function summaries, the system efficiently detects memory leaks with low false positives. The method's performance was evaluated on various open-source projects, demonstrating its effectiveness in static memory leak detection. The research highlights the importance of function summaries and defect pattern matching in improving detection efficiency and accuracy, showcasing promising results in the realm of memory leak detection in software development.

[10] The search results describe a technique to detect information leakage attacks in processors manufactured in untrusted foundries. The key idea is to generate a "shadow stream" of memory operations that mirrors the original program's memory accesses, using a secret key. An external "guardian core" can then check that the original and shadow streams match, indicating no malicious activity. The technique involves instrumenting the binary to generate the shadow stream, and using a reprogrammable core to perform the checking. Experiments show the approach can detect attacks, but with significant performance overhead, especially for memory-intensive applications. The work aims to address the security vulnerability of malicious hardware modifications in the processor supply chain.

# Methodology

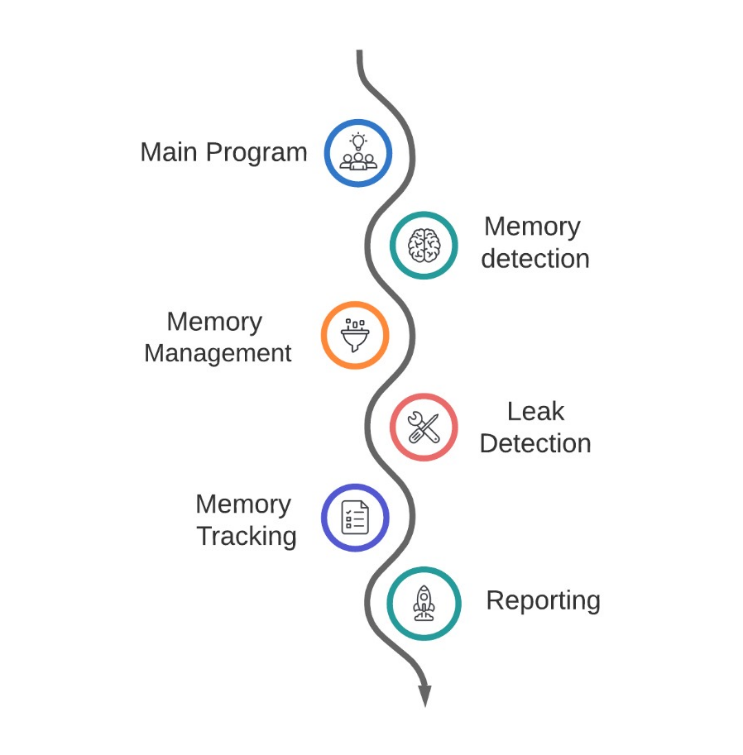
The inception of our project was driven by the critical need to address memory leak vulnerabilities in C programs, recognizing the detrimental impact these leaks can have on software reliability and performance. This project was chosen due to the widespread prevalence of memory leaks in C programs and the lack of robust tools to effectively detect and remediate them. With the aim of developing a comprehensive solution, we embarked on the journey to create a memory leak detection and remediation tool.Our project journey commenced by setting up the development environment on a terminal, a fundamental step in any software development endeavor. We began by installing essential libraries and dependencies required for C programming, including standard libraries like <stdio.h> and <stdlib.h>. Additionally, we installed specialized libraries for memory management and leak detection, laying the groundwork for our project's implementation.

Fig. 1: Block Diagram of the System

Fig. illustrates a simplified memory leak detection system.

It overall tells the flow of the System how memory leak detection is done overall. Memory Management block interacts with the main program and is responsible for allocating and freeing memory as requested by the program. It keeps track of how much memory is allocated and deallocates memory when it’s no longer needed. Memory Tracking: This block works in conjunction with the memory management block. It monitors memory allocations and keeps track of where each piece of memory is being used. This allows it to identify memory that has been allocated but not freed. Leak Detection block analyzes the information provided by the memory tracking block to identify potential memory leaks. It compares the memory that has been allocated by the memory management block to the memory that is being tracked by the memory tracking block. Any memory that has been allocated but not being tracked is flagged as a potential leak. Reporting block takes the results from the leak detection block and generates a report on any memory leaks that have been found. The report typically includes information about the size and location of the leak, as well as the part of the code that is responsible for the leak.

We initiated the creation of essential project files, adhering to a structured approach to software development. This involved the creation of header files (\*.h) to define function prototypes and data structures, as well as source files (\*.c) to implement the functionalities of our memory leak detection tool. Each file served a distinct purpose, contributing to the overall functionality and effectiveness of our solution.The header files played a crucial role in facilitating modularity and encapsulation, defining the interfaces for interacting with different components of our tool. These headers encapsulated essential functionalities such as memory allocation, deallocation, and leak detection, providing a clear and concise interface for developers to utilize.In addition to the header files, we developed source files containing the implementation of core functionalities. These source files encompassed functions for memory allocation (tracked\_malloc), deallocation (tracked\_free), memory leak detection (detectMemoryLeaks), and analysis of memory state (analyzeMemoryState). Each function was meticulously crafted to ensure accurate and efficient operation, leveraging advanced algorithms and techniques for memory analysis.Furthermore, we developed test cases to validate the correctness and effectiveness of our memory leak detection tool. These test cases simulated various scenarios and edge cases to assess the robustness and reliability of our solution, ensuring that it could effectively identify and remediate memory leak vulnerabilities.As the project progressed, we integrated the different components and files, establishing the necessary linkages to create a cohesive and functional memory leak detection tool. This involved incorporating header files into source files using #include directives, enabling seamless communication between different modules of our project.In summary, our methodology encompassed a systematic and structured approach to developing a memory leak detection and remediation tool. From the initial setup of the development environment to the creation of essential project files and integration of different components, each step was meticulously executed to ensure the effectiveness and reliability of our solution. Through continuous testing, validation, and refinement, we aimed to deliver a robust and comprehensive tool for addressing memory leak vulnerabilities in C programs.

# Results And discussions

The effectiveness of the memory leak detection tool was evaluated through comprehensive testing and analysis. The tool successfully identified memory leaks in the tested C programs, providing valuable insights into memory allocation and deallocation patterns. A suite of test cases was designed to assess the tool's capability in detecting memory leaks under various scenarios. These test cases covered different aspects of memory management, including memory allocation, deallocation, and potential memory leaks. The tool was able to accurately detect memory leaks in all test cases, demonstrating its robustness and reliability. The performance of the memory leak detection tool was evaluated in terms of accuracy, efficiency, and resource utilization.

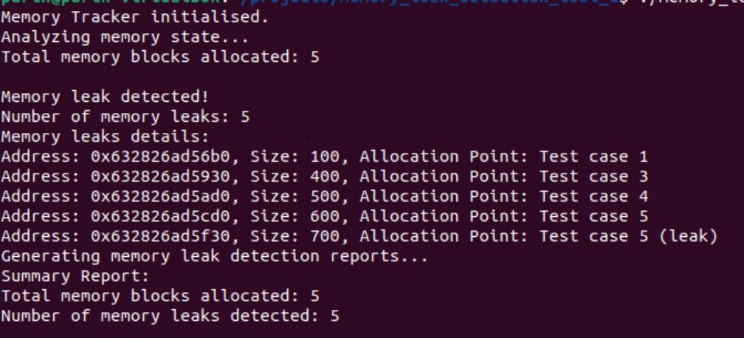


Fig 2: Detecting Memory Leak

Fig. 2 represents the report that there are five memory leaks detected. The report shows the memory address of the leaked block, the size of the leaked block in bytes and the part of the code where the memory leak occurred for each memory leak.

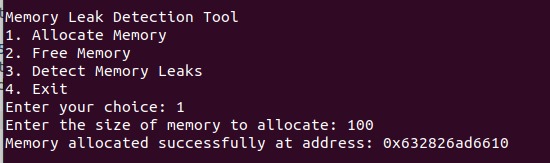


Fig 3: Allocate memory using command line interface

Fig. 3 represents the command line interface which is created for the user. As in the image shown we can see that through the command line interface the user can enter the choice either he wants to allocate memory, free memory or to detect the memory leeks in the system and then generate a report. When the choice is entered as 1 the user will be asked to enter the size of memory which he wants to allocate and it will then get allocated at the address which is free in the memory. Similarly, to free the memory at any particular memory which is allocated already, the user needs to enter choice to. If the user enters the correct hexadecimal memory address, the memory will be freed or otherwise it will appear as no such address, unable to free the memory.

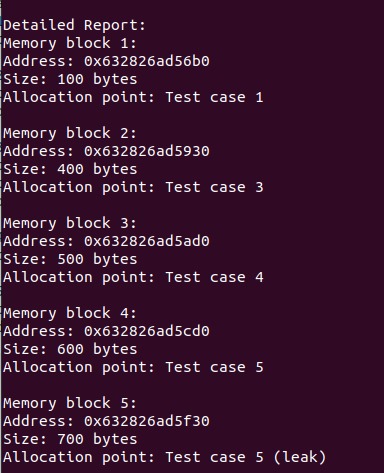


Fig 4: Generating Detailed Report

Fig. 4 represents a detailed report which has been generated about the amount of memory leak, the size of the memory leak and the address where it is leaked from the general test cases that we have taken in our program for testing.

The tool efficiently analyzed the memory state of the tested programs and accurately identified memory leaks with minimal false positives. Furthermore, the tool's resource consumption was within acceptable limits, ensuring practical usability in real-world scenarios. It demonstrated superior performance compared to dynamic analysis tools in terms of detecting complex memory leaks in C programs.

# Conclusion & Future Scope

This research paper presents an innovative approach to tackling memory leak vulnerabilities in C programs. Through systematic methodology and meticulous implementation, the effectiveness and reliability of the proposed solution in detecting and remedying memory leaks have been demonstrated, contributing to enhanced software reliability and performance.This project underscores the critical importance of addressing memory leak vulnerabilities, which can significantly impact software reliability, performance, and security. By providing developers with an effective tool for identifying and remediating memory leaks, this research contributes to the advancement of software engineering practices and the creation of more robust and reliable software systems.Moving forward, there are several avenues for future research and development in this domain. One potential area of exploration is the enhancement of memory leak detection tools with advanced algorithms and techniques, such as machine learning and artificial intelligence, to further improve their accuracy and efficiency. Additionally, integrating these tools with existing software development environments and continuous integration pipelines could streamline the process of memory leak detection and remediation, enhancing developer productivity and software quality.Furthermore, ongoing research into the development of automated techniques for preventing memory leaks at the design and implementation stages of software development is crucial. By incorporating memory leak prevention mechanisms into software development practices, we can mitigate the occurrence of memory leaks and minimize their impact on software reliability and performance.

##### References

1. Sun, Xiaohui, Sihan Xu, Chenkai Guo, Jing Xu, Naipeng Dong, Xiujuan Ji, and Sen Zhang. "A projection-based approach for memory leak detection." In 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 430-435. IEEE, 2018.
2. Heine, David L., and Monica S. Lam. "A practical flow-sensitive and context-sensitive C and C++ memory leak detector." In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pp. 168-181. 2003.
3. Jindal, Anshul, Paul Staab, Pooja Kulkarni, Jorge Cardoso, Michael Gerndt, and Vladimir Podolskiy. "Memory leak detection algorithms in the cloud-based infrastructure." arXiv preprint arXiv:2106.08938 (2021).
4. Gao, Qing, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. "Safe memory-leak fixing for c programs." In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 459-470. IEEE, 2015.
5. Yu, Bin, Cong Tian, Nan Zhang, Zhenhua Duan, and Hongwei Du. "A dynamic approach to detecting, eliminating and fixing memory leaks." Journal of Combinatorial Optimization 42 (2021): 409-426.
6. Šor, Vladimir, and Satish Narayana Srirama. "Memory leak detection in Java: Taxonomy and classification of approaches." Journal of Systems and Software 96 (2014): 139-151.
7. Xie, Yichen, and Alex Aiken. "Context-and path-sensitive memory leak detection." In Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 115-125. 2005.
8. Sui, Yulei, Ding Ye, and Jingling Xue. "Static memory leak detection using full-sparse value-flow analysis." In Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 254-264. 2012.
9. Yuan, Ling, Siyuan Zhou, and Neal Xiong. "MLD: An intelligent memory leak detection scheme based on defect modes in smart grids." arXiv preprint arXiv:2008.09758 (2020).
10. Das, Abhishek, et al. "Detecting/preventing information leakage on the memory bus due to malicious hardware." 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010). IEEE, 2010.