

# Assessing Software Weakness Detection Capabilities: An Empirical Study of IKOS

Hripsime Hovhannisyan, Gayane Mirakyan, Edgar Khachatryan, Hayk Aslanyan

Center of Advanced Software Technologies, Russian-Armenian University, Yerevan, Armenia

hripsime.hovhannisyan@rau.am, mirakyan.gayane@rau.am, khachatryan.edgar@rau.am, hayk.aslanyan@rau.am

**Abstract**—Memory-related vulnerabilities, such as use-after-free and double-free errors, remain a critical security issue in C and C++ programs, potentially leading to arbitrary code execution and data corruption. While dynamic analysis offers high precision, it often incurs significant overhead; conversely, static analysis provides broader coverage but faces challenges with false positives and complex memory modeling. This paper evaluates the capabilities of IKOS, a static analysis tool developed by NASA, in detecting these memory errors. We assess IKOS using the standardized Juliet Test Suite, focusing on its handling of flow, context, and path sensitivity. Our results indicate that IKOS achieves high detection rates, with an average F1 score of 95.46% for double-free and 89.67% for use-after-free scenarios. However, the analysis reveals limitations in IKOS's ability to handle complex C++ containers and global variables. This study details the specific coding patterns that cause false negatives—such as wide character usage and widening techniques in loops—providing insights for practitioners selecting the IKOS analysis tool and researchers improving static analysis precision.

**Keywords**—Static Analysis, IKOS, Memory Vulnerabilities, Use-After-Free, Double-Free, Juliet Test Suite.

## I. INTRODUCTION

Memory-related vulnerabilities remain a critical and widespread issue in software development, particularly within languages like C and C++ that rely on manual memory management. Common errors such as use-after-free, double-free, buffer overflows, and memory leaks can have severe security implications, potentially leading to denial-of-service attacks, information theft, or arbitrary code execution.

To mitigate these risks, developers employ various detection methods, primarily categorized into dynamic and static analysis[1]. While dynamic analysis tools (e.g., Valgrind[2][3], Dr.Memory[4] and fuzzing tools[5][6][7]) monitor execution at runtime to provide high precision, they often suffer from significant overhead and incomplete code coverage. In contrast, static analysis examines source code without execution, offering broader coverage and earlier detection in the development cycle, though often with the trade-off of higher false positive rates and difficulties in modeling complex memory behaviors.

This work focuses on evaluating the capabilities of IKOS[8][9], a static analysis tool designed to detect memory-related errors in C/C++ programs. We selected IKOS for this study because it relies on Abstract Interpretation, a mathematical framework that guarantees soundness—meaning

it theoretically should not miss errors if modeled correctly. Unlike heuristic-based tools that prioritize speed over accuracy, IKOS is designed for high-assurance domains (e.g., NASA avionics), making it a critical candidate for evaluating precise memory safety. We assess IKOS's effectiveness in identifying use-after-free and double-free vulnerabilities using the Juliet Test Suite[10], a standardized benchmark for memory error detection. We skipped evaluation for memory leaks, as it was discussed in our previous work[11]. The evaluation aims to determine the tool's practical utility and precision in handling complex memory management scenarios.

The remainder of this article is structured as follows: Section II details the nature of memory-related errors and their security impact. Section III introduces the IKOS tool and its architectural features. Section IV discusses critical static analysis properties, such as flow, field, and context sensitivity, that influence detection accuracy. Section V presents the evaluation results of IKOS on the Juliet Test Suite. Finally, Section VI concludes with a summary of findings and recommendations for practitioners and researchers.

## II. NATURE OF MEMORY-RELATED ERRORS AND SECURITY IMPACT

Memory-related errors occur when a program incorrectly manages memory allocation, access, or deallocation. In languages that require manual memory management (such as C and C++), these errors constitute a significant portion of modern software vulnerabilities.

This section details the nature of these errors and their direct security implications.

### A. Use After Free

Use-After-Free vulnerabilities arise when a program continues to use a pointer to a memory location after that memory has been freed (deallocated).

1. **Allocation:** The program allocates memory and sets a pointer (e.g., *ptr*) to reference it.
2. **Free:** The program frees the memory, effectively marking it as "available" for the system to reuse.
3. **Dangling Pointer:** The program fails to clear *ptr* (set it to NULL). *ptr* still points to the old address.

4. **Reuse:** The system reallocates that memory address for a new object/data.
5. **Illegal Access:** The program mistakenly uses the old ptr to write to that address, corrupting the *new*, valid object that now lives there.

### Security Impact

- **Data Corruption and Logic Bypass:** An attacker can manipulate the heap layout so that sensitive data (like a user permission object) lands in the freed spot. The dangling pointer is then used to modify this new object, potentially escalating privileges (e.g., changing *isAdmin = false* to *true*).
- **Code Execution:** If the new object contains a function pointer (common in C++ vtables), overwriting it via the dangling pointer can redirect execution flow to malicious code.

### B. Double Free

This error occurs when `free()` is called on the same memory address more than once.

1. Pointer *ptr* is freed.
2. The memory manager adds this chunk to its "free list" (a list of available spots).
3. Pointer *ptr* is freed again.
4. The memory manager may corrupt its own internal list, creating loops or linking the same chunk twice.

**Heap Corruption/Write-What-Where:** When the program later requests memory, the allocator might return the same memory address for two different objects. This allows the attacker to overwrite the data of one object by modifying the other, leading to similar exploitation paths as Use-After-Free (privilege escalation or code execution).

## III. IKOS DESCRIPTION

IKOS is an open-source static analyzer developed by NASA for the certification of avionic systems. It provides a generic and efficient implementation of advanced abstract interpretation techniques[12], including data structures and algorithms such as control-flow graphs, fixpoint iterators, and numerical abstract domains. In addition, IKOS comes with a C and C++ static analyzer built on LLVM[13][14], which can efficiently check programs to detect and prove the absence of runtime errors.

IKOS uses dynamic inlining, analyzing each function only once and creating a reusable summary for later use. Furthermore, IKOS offers a mathematical guarantee that all computed properties hold across all possible execution paths of the program. Its analysis is interprocedural, which means it looks across multiple functions. At first, each function is analyzed without considering the specific context of its calls. Later, full context-sensitivity is reached by inlining all program functions, allowing a more precise expansion of function calls.

The IKOS can analyze both a source file (.c, .cpp) and an LLVM bitcode file (.bc) as input. After analyzing the code, it

detects potential runtime errors (such as undefined behaviors), generates a result database named `output.db` in the current directory, and produces a detailed report. Moreover, IKOS provides several checks, including double-free analysis, which helps detect issues such as double-free, invalid-free, use-after-free, and use-after-return errors.

## IV. STATIC ANALYSIS PROPERTIES AND THEIR IMPACT ON MEMORY ERROR DETECTION

The effectiveness of static analysis tools in detecting memory-related errors depends critically on several fundamental properties that determine their analysis precision and scope. These properties - flow sensitivity, field sensitivity, context sensitivity, path sensitivity, and the distinction between interprocedural and intraprocedural analyses directly influence a tool's ability to track memory errors, distinguish true vulnerabilities from false alarms, and handle real-world complex code patterns. Understanding these properties is essential for evaluating tool capabilities and selecting appropriate analyzers for specific use cases. In this section, we examine each property and explain why it matters for memory error detection.

### A. Flow Sensitivity

A flow-sensitive analysis considers the order of statements in the program. In contrast, a flow-insensitive analysis merges information from all statements regardless of execution order. Flow sensitivity is critical for detecting use-after-free and double-free errors, because these vulnerabilities depend on the specific sequence of memory operations. A flow-insensitive analysis can not distinguish between a pointer that has been freed and subsequently dereferenced (a bug) versus a pointer that is dereferenced before being freed (correct behaviour).

### B. Field Sensitivity

A field-sensitive analysis distinguishes between different fields of a struct or class. Field-insensitive analysis treats the entire struct as a single unit.

In C/C++, structs often contain multiple pointer fields, each with independent allocation states. Memory errors may affect one field while others remain valid. Field-insensitive analysis cannot distinguish which field has the error.

### C. Context sensitivity

A context-sensitive analysis distinguishes between different calling contexts when analyzing a function. It maintains separate abstract states for each call site, recognizing that the same function may behave differently depending on how it is called. Context-insensitive analysis, by contrast, merges all calling contexts into a single analysis.

Context sensitivity is critical when memory management functions are user-defined functions or are abstracted as helper functions, or when the same function behaves differently

based on the value of some argument or global or environment and is called with different allocation and deallocation sites.

#### D. Path sensitivity

Path-sensitive analysis distinguishes between different execution paths through the program, maintaining separate states for each feasible path. Path-insensitive analysis merges states from all paths at control-flow join points. Memory operations are often under conditional checks. Path sensitivity enables the analyzer to understand that certain error conditions are impossible on specific execution paths.

#### E. Intraprocedural and Interprocedural analyses capability

Interprocedural analysis is a capability that allows for the tracking of memory access and changes across function calls, considering the behavior of called functions. Without this capability, an analyzer might report false positives or, conversely, miss actual errors. While interprocedural analysis offers greater accuracy, it is computationally expensive. It can sometimes require analyzing the same function multiple times, which may lead to scalability issues.

#### F. IKOS Properties

We attempted to contact the tool contributors to clarify specific sensitivities, but **received no response**. As a result, we utilized small examples from [15] to empirically examine the tool. Our evaluation indicates that IKOS satisfies all the required properties.

### V. IKOS RESULTS ON THE JULIET TEST SUITE

To evaluate how well IKOS performs, we built an automated testing system using the Juliet Test Suite version 1.3. This system automatically runs IKOS version 3.5 against the code and calculates its accuracy scores: True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). We also manually checked the mistakes IKOS made (specifically the false positives and false negatives). This helped us identify exactly where IKOS struggles and suggest ways to improve it in the future.

#### A. Juliet Test Suite

The Juliet Test Suite is a systematic collection of synthetic test programs developed by the National Security Agency's Center for Assured Software. Designed to evaluate the capabilities of software assurance tools—particularly static analyzers—the suite targets **C/C++ and Java** environments.

It covers **over 100 distinct classes of errors**, including common vulnerabilities such as memory leaks, use-after-frees, double-frees and OS injections. The suite contains more than 80,000 test cases. Each test case includes both "good" and "bad" functions. Good functions represent correct code without the specified vulnerability, while bad functions intentionally contain the target bug. In the Juliet Test Suite,

bad functions are identified by naming conventions: they end with "\_bad" in C and "Bad" in C++ test cases.

For our evaluation, we integrated two test sets from the Juliet Test Suite, focusing on memory-related errors:

- **CWE-415:** Double-free errors (962 test cases)
- **CWE-416:** Use-after-free errors (459 test cases)

For the tool, we verified whether it correctly reported the specified issue in bad functions (true positives) and refrained from reporting issues in good functions (true negatives). The complete results for all evaluated tests are presented in TABLE 1.

#### B. Results Description

The performance evaluation of IKOS on the Juliet Test Suite is detailed in Tables 1 through 5. A standout characteristic of the results was the tool's perfect precision (100%) across all tested categories in both C and C++. In every instance where IKOS flagged a vulnerability, it was a confirmed true positive, with zero false positives (FP) recorded across 1,421 total test cases.

However, this strict error reporting behavior resulted in missed detections (false negatives). While IKOS achieved a robust 94.74% recall for Double Free errors in C, it demonstrated reduced sensitivity for Use After Free vulnerabilities. The lowest performance was observed in the Use After Free (C) category, where the recall dropped to 78.26%, corresponding to 30 missed vulnerabilities. It is important to note that we applied a strict evaluation strategy: we only counted explicit 'Errors' and excluded inconclusive 'Warnings'. Under these terms, IKOS is highly reliable; when it classifies a bug as an Error, it is accurate. However, in practice, the tool also outputs a significant number of lower-priority warnings which we excluded from these statistics. IKOS received a 95.46% and 89.67% average F1 score for double free and use after free, respectively.

TABLE 1. CONFUSION MATRIX DOUBLE FREE C

| Actual Class                     | Predicted Positive | Predicted Negative |
|----------------------------------|--------------------|--------------------|
| <b>Positive (Vulnerable)</b>     | 216 (TP)           | 12 (FN)            |
| <b>Negative (Not Vulnerable)</b> | 0 (FP)             | 228 (TN)           |

TABLE 2. CONFUSION MATRIX DOUBLE FREE C++

| Actual Class                     | Predicted Positive | Predicted Negative |
|----------------------------------|--------------------|--------------------|
| <b>Positive (Vulnerable)</b>     | 646 (TP)           | 88 (FN)            |
| <b>Negative (Not Vulnerable)</b> | 0 (FP)             | 734 (TN)           |

TABLE 3. CONFUSION MATRIX USE AFTER FREE C

| Actual Class | Predicted Positive | Predicted Negative |
|--------------|--------------------|--------------------|
|--------------|--------------------|--------------------|

|                                      |          |          |
|--------------------------------------|----------|----------|
| <b>Positive<br/>(Vulnerable)</b>     | 108 (TP) | 30 (FN)  |
| <b>Negative<br/>(Not Vulnerable)</b> | 0 (FP)   | 138 (TN) |

TABLE 4. CONFUSION MATRIX USE AFTER FREE C++

|                                      |                           |                           |
|--------------------------------------|---------------------------|---------------------------|
| <b>Actual Class</b>                  | <b>Predicted Positive</b> | <b>Predicted Negative</b> |
| <b>Positive<br/>(Vulnerable)</b>     | 271 (TP)                  | 50 (FN)                   |
| <b>Negative<br/>(Not Vulnerable)</b> | 0 (FP)                    | 321 (TN)                  |

TABLE 5. IKOS RESULTS ON JULIET TEST SUITE

|                             |                  |               |                 |
|-----------------------------|------------------|---------------|-----------------|
| <b>Category</b>             | <b>Precision</b> | <b>Recall</b> | <b>F1 score</b> |
| <b>C - Double Free</b>      | 100.00%          | 94.74%        | 97.30%          |
| <b>C - Use After Free</b>   | 100.00%          | 78.26%        | 87.80%          |
| <b>C++ - Double Free</b>    | 100.00%          | 88.01%        | 93.62%          |
| <b>C++ - Use After Free</b> | 100.00%          | 84.42%        | 91.55%          |

IKOS struggled with specific coding patterns due to how it tracks memory and guesses program behavior:

- **Unrecognized Functions (wchar\_t):** The tool consistently missed bugs involving wide characters (like `wchar_t`) when used with `wprintf`. This happened because IKOS likely doesn't have a "definition" for `wprintf` in its internal library. It simply didn't know that `wprintf` accesses the memory pointer, so it assumed the code was safe.
- **Global Variables:** IKOS failed when the bug was hidden inside an if statement controlled by a global variable. The tool couldn't figure out the value of the global variable during analysis, so it likely decided that the risky code path was "unreachable" and ignored it.
- **Multiple Loops:** IKOS failed when the memory was freed in one loop and used in a different loop. This failure comes from IKOS's use of fixpoint iterators and widening operators to handle cyclic control flow. To prevent infinite analysis time, the analyzer is generalized. In this process, the precise history of the pointer (specifically that it was definitively freed in the first loop) is often over-generalized to *Top*[12]. Once the memory state reaches *Top*, it represents complete uncertainty—meaning the analyzer considers the pointer as potentially being in any state (valid, freed, or unallocated). Consequently, when the second loop attempts to free or use the pointer again, IKOS cannot prove with certainty that the memory was already freed. As the tool typically issues a warning rather than a definite error, this scenario resulted in a False Negative under our strict evaluation criteria, which classify only definitive errors as True Positives.

- **Complex Containers (Vectors/Maps):** The tool failed to track pointer lifetimes within complex structures like `std::vector` or `std::map` due to the complex control flow inherent in STL implementations. Standard library functions (e.g., `push_back` or element insertion) rely internally on loops, conditional logic, and external system calls to manage memory. To handle these internal loops without entering infinite recursion, IKOS applies widening techniques. This later causes a similar issue described in the multiple loops scenario.

## VI. CONCLUSION

This study presented a comprehensive evaluation of the IKOS static analyzer, focusing on its effectiveness in detecting critical memory vulnerabilities in C and C++ applications. By utilizing the Juliet Test Suite, we demonstrated that IKOS provides robust detection capabilities for double-free and use-after-free errors, satisfying key analysis properties such as flow, context, and path sensitivity.

Despite its strong performance, our empirical analysis uncovered specific limitations. IKOS faces challenges when analyzing complex C++ Standard Template Library (STL) containers such as vectors and maps, where it often loses track of pointer lifecycles. Furthermore, the tool exhibited false negatives in scenarios involving global variable-dependent control flows and wide character functions (`wchar_t`), likely due to missing internal definitions and aggressive widening techniques applied during loop analysis.

These findings highlight the trade-offs between analysis scalability and precision. Future work should focus on enhancing IKOS's modeling of C++ container internals and refining its widening strategies to improve accuracy across multiple loops. For developers, IKOS remains a valuable tool for certification-grade analysis, provided that its known blind spots regarding complex data structures are taken into account.

## ACKNOWLEDGMENT

The research was supported by the Higher Education and Science Committee of MESCS RA (Research project No 24AA-1B016) and funded by the Analysis Research & Planning for Armenian (ARPA) Institute.

## REFERENCES

- [1] A. Aho, J. Ullman, R. Sethi, and M. Lam, Compilers: Principles, Techniques, and Tools. Boston, MA, USA: Addison Wesley, 2006, p. 1040.
- [2] Valgrind. Accessed: Dec. 5, 2025. [Online]. Available: <https://valgrind.org/>.
- [3] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in Proc. 28th ACM SIGPLAN Conf. Program. Lang. Design Implement., Jun. 2007, pp. 89–100.
- [4] Dr.Memory. Accessed: Dec. 5, 2025. [Online]. Available: <https://github.com/dynamorio/drmemory>.
- [5] AFL. Accessed: Dec. 5, 2025. [Online]. Available: <https://github.com/google/AFL>.
- [6] LibAFL. Accessed: Dec. 5, 2025. [Online]. Available: <https://github.com/AFLplusplus/LibAFL>.
- [7] S. Sargsyan, S. Kurmangaleev, J. Hakobyan, M. Mehrabyan, S. Asryan and H. Movsisyan, "Directed Fuzzing Based on Program Dynamic

- Instrumentation," 2019 International Conference on Engineering Technologies and Computer Science (EnT), Moscow, Russia, 2019, pp. 30-33.
- [8] G. Brat, J. A. Navas, N. Shi, and A. Venet, "IKOS: A Framework for Static Analysis based on Abstract Interpretation," in *Proceedings of the 12th International Conference on Software Engineering and Formal Methods (SEFM)*, Grenoble, France, 2014, pp. 271–277.
  - [9] IKOS. Accessed: Dec. 5, 2025. [Online]. Available: <https://github.com/NASA-SW-VnV/ikos>.
  - [10] Juliet Test Suite. Accessed: Dec. 5, 2025. [Online]. Available: <https://samate.nist.gov/SRD/testsuite.php>.
  - [11] H. Aslanyan, Z. Gevorgyan, R. Mkoyan, H. Movsisyan, V. Sahakyan, and S. Sargsyan, "Static analysis methods for memory leak detection: A survey," in *Proc. Ivannikov Memorial Workshop (IVMEM)*, Moscow, Russia, Sep. 2022, pp. 1–6.
  - [12] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1977, pp. 238–252.
  - [13] The LLVM Compiler Infrastructure. Accessed: Dec. 5, 2025. [Online]. Available: <https://llvm.org>.
  - [14] Whole Program LLVM IR in Go. Accessed: Dec. 5, 2025. [Online]. Available: <https://github.com/SRI-CSL/gllvm>.
  - [15] GSAC. Accessed: Dec. 5, 2025. [Online]. Available: [https://github.com/GSACTech/contest/tree/main/resources/test\\_suites/sensitivities](https://github.com/GSACTech/contest/tree/main/resources/test_suites/sensitivities).

**Hripsime Hovhannisyan** received her B.Sc. and M.Sc. degrees in informatics and applied mathematics from Yerevan State University,

Armenia, in 2019 and 2021, respectively. With nearly six years of experience in software development, she specializes in security, demonstrating proficiency in both code static and dynamic analysis. Her research interests include graph theory, software security, and software testing. She has been pursuing a Ph.D. degree at the Russian-Armenian University since 2024.

**Gayane Mirakyan** received the B.Sc. degree in informatics and applied mathematics from Russian–Armenian University, Armenia, in 2024, where she is currently pursuing the M.Sc. degree in system programming. In 2021, she joined the Center of Advanced Software Technologies, as a Laboratory Assistant. Her main responsibilities in this role involve the development and testing of static analysis tools. Her research interests include software engineering, software static analysis, and software testing.

**Edgar Khacatryan** is currently pursuing the bachelor's degree in informatics and applied mathematics with Russian–Armenian University. In 2024, he joined the Center of Advanced Software Technologies, as a Laboratory Assistant. His main responsibilities in this role involve the development and testing of static analysis tools. His research interests include software engineering, software static analysis, and software testing.

**Hayk Aslanyan** received the B.Sc. and M.Sc. degrees in informatics and applied mathematics from Yerevan State University, Armenia, graduating in 2014 and 2016, respectively. He obtained a Ph.D. degree in 2019 in mathematical and software support for computing machines, complexes, and computer networks from Ivannikov Institute for System Programming, Russia. Currently, he is a lecturer and researcher at the Russian-Armenian University, Armenia. His research interests include program static and dynamic analysis, software security, and compiler optimization.