Refined Traces for Root Cause Analysis in Heap Buffer Overflow Detection

Blake Hinkley, Ben Sippel

CSEC 759

3 December 2025

Abstract

Heap-based buffer overflows remain a significant source of security vulnerabilities. Static analysis tools designed to detect such flaws face tradeoffs between precision, scalability, and soundness, and developers are often left with reports that are often difficult to understand and diagnose the issue. This work presents a refinement approach for traces generated by a heap-disjointness-based static analyzer, transforming sparse, repetitive outputs into robust, self-contained artifacts. By adding contextual information, removing redundant data, and ensuring correct alignments with the source code, the refined traces enable faster and more accurate root-cause analysis, improving developer interpretability without substantial computational overhead.

Background

The Open Worldwide Application Security Project (OWASP) defines a buffer overflow as a vulnerability that occurs when a program “attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer”. A heap-based buffer overflow specifically arises when a buffer allocated on the heap using routines such as `malloc()` is overwritten, potentially corrupting adjacent memory and leading to undefined behavior or security breaches. Detection of heap-based buffer overflows relies on precise modeling of memory structures and aliasing relationships. Over the past decade, symbolic execution and fuzzing approaches have advanced detection capabilities. For example, Li et al. (2010) demonstrated that symbolic analysis can practically identify buffer overflows, though it

remains constrained by path explosion and scalability challenges in large or complex software.Google's OSS-Fuzz, launched in 2016, automates fuzzing for large-scale open-source projects, uncovering thousands of bugs and security vulnerabilities; its limitations include incomplete path coverage and inability to detect logic-dependent overflows in code paths that are difficult to reach. The heap-disjointness assumption, where distinct symbolic heap accesses are presumed non-aliased unless syntactically identical, improves scalability and practical precision at the cost of formal soundness (Guo et al., 63-75).

Memory / Overflow Type	***Memory Model for Static Analysis (2010)***	***Empirical Study (2016)***	***Heap Disjointness Detector (2024)***
Heap buffer overflow	Fully supported via symbolic regions	Studied as part of the dataset	Primary focus (core detection target)
Stack buffer overflow	Supported via region modeling of locals	Studied as part of the dataset	**Not supported**
Global/static buffer overflow	Supported (global regions modeled)	Studied as part of the dataset	**Not supported**
Array index out-of-bounds	Fully supported	Studied as part of the dataset	If array is on heap
Pointer arithmetic issues	Modeled in region offsets	Studied in depth from results	Central to symbolic access paths (disjunction)
Root Cause Analysis	**Not addressed**	Entirely human analysis	**Not addressed**

Figure 1: Related Work Direct Comparisons

Trace-based root-cause analysis (RCA) methods are increasingly applied to assist developers in diagnosing errors. The faster a team can identify the important portions of a detection, the faster they can appropriately respond to it and move forward. There are different formulations of robust traces, the formulation our project decided to use as guidance was the one proposed in Zhang et al. (2024). Figure 2 displays the primary characteristic of their trace logs, in which a trace is made of spans with spans containing relevant information for traversal and analysis. This formulation made it easier for the researchers to automate root cause analysis, and leveraged this strong baseline trace formulation for the creation of their own tool called TraceContrast.

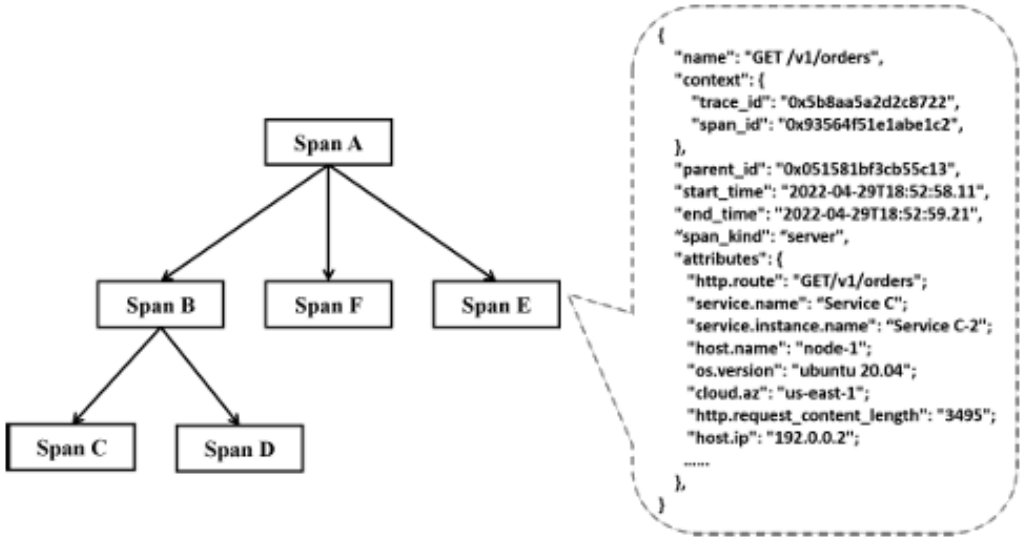

Figure 2: An example of Trace and Span Log

Baseline

This research builds on two primary foundations that together establish both the detection capability and the trace-structuring framework required for effective root-cause analysis of heap-based buffer overflows.

- **Heap-disjointness analyzer (2024) -** This tool employs symbolic execution under the heap-disjointness assumption, which treats distinct symbolic heap accesses as non-aliased unless syntactically identical. This design choice enhances scalability and allows practical precision when analyzing large or complex software, making it particularly suitable for heap overflow detection. However, the analyzer generates verbose traces that are often sparse in meaningful information, including repeated identifiers across lines with multiple findings. Occasionally, the trace contains misaligned references due to discrepancies between bitcode and source-level metadata. These characteristics hinder straightforward interpretation, requiring significant effort from developers to reconstruct the causal sequence of the overflow.
- **Multi-dimensional RCA framework (2024) -** The Trace formulation displayed within this paper was very effective for later ingesting by other tooling. The TraceContrast tool developed in this study is able to localize multi-dimensional root causes efficiently through the careful sequence constructed in a trace. The initial trace created by the heap-disjointness analyzer has its own sequence structure where it will give the steps it took to reach a possible heap buffer overflow. This sequential flow is similar to the one seen within this paper's framework, with the key difference being that spans have much more information in them than the steps provided by the heap-disjointness analyzer. As such we decided that including important execution information, such as the line of code referenced by the analyzer, would help create more robust traces that are more in-line with modern RCA techniques and thus more useful to developers.

Methodology

The refinement process is designed to transform verbose, sparse traces from the baseline heap-disjointness analyzer into coherent, developer-ready artifacts suitable for root-cause analysis. The process evaluates each trace step for its contribution to reconstructing causal paths while systematically filtering noise that obscures meaningful information. The methodology consists of four primary components:

1. **Value Identification -** Each trace line is assessed for its relevance to understanding the overflow's cause. Steps that provide critical information about memory accesses, variable states, or control-flow decisions are retained, while those with little interpretive value are removed. This ensures that developers can follow the sequence of events leading to the overflow without sifting through extraneous data.
2. **Redundancy Reduction -** Traces often include repeated metadata, such as file and function identifiers, or multiple symbolic representations of the same operation. These duplications are collapsed or removed to streamline the trace, reducing cognitive load without discarding essential causal information. This step improves readability and prevents developers from being misled by repetitive entries.
3. **Misalignment Correction -** Bitcodes files are not guaranteed to store line information in the exact same way across systems or across different LLVM versions. We observed that occasionally there would be some form of 'drift' where a noticeable difference exists between the line reported in the initial trace and the actual line where problematic code surfaces. To remedy this without

necessarily relying on specific LLVM versions of .bc files, and thus make the tool solely require the initial trace and initial code, the trace's behavior was leveraged. Any branch in execution, such as an if statement, would be logged by the tool since it tracks expectant execution towards the overflow. This means that any if statement en-route to the detected bug would be recorded, so there's effectively always at least one branch statement within any given trace formulation. Any function that does not have a single branch is likely minor enough to not have any misalignment in the first place. There are only so many forms of branch statements within the C programming language, leveraging these 2 known factors it becomes possible to calculate the exact misalignment without ever having to refer back to the bitcode files after the initial analyzer has been executed.

4. **Contextual Enhancement -** To further support comprehension, relevant source lines are embedded with the trace. Care is taken to provide sufficient context without introducing excessive verbosity that could result in a bloated report. This step enhances interpretability by situating each symbolic step within the broader program logic.

The overall methodology is guided by iterative exploration of baseline traces, manual inspection, and reconstruction of improved equivalents. By applying these steps systematically, the process generates self-sufficient traces that are immediately interpretable, significantly reducing the effort required for root-cause analysis while preserving the detailed information necessary for accurate diagnosis.

Results

The refinement methodology was evaluated on two representative codebases, tcpdump and zstd. These were selected because the provided Docker image included their bitcode representations, and had quick execution time of the original static analysis tool while offering a diverse set of test cases for evaluating the refined traces.

- **Processing Efficiency -** Despite the additional steps involved in refinement, including redundancy reduction, misalignment correction, and contextual enhancement, the overall processing overhead remained minimal. On average, refining a single trace required less than one second of additional computation. This demonstrates that the methodology is practical for real-world use, where large codebases may generate hundreds or thousands of traces, and performance considerations are critical for integration into automated pipelines.
- **Improved Usability -** Refined traces exhibited significantly enhanced readability and interpretability. Metadata redundancy, such as function or file identifiers, was collapsed, and misaligned symbolic entries were corrected to match source-level offsets. Additionally, the inclusion of relevant source lines provided developers with immediate contextual understanding, reducing the need to cross-reference multiple files or manually inspect the original code. In practice, these improvements allow analysts to reconstruct the casual chain of heap buffer overflows more accurately and with far less effort than with baseline traces.
- **Limitations -** Developing resilient refinement across edge cases proved challenging. Semi-automated analysis often produced traces with excessive information, limiting efficiency gains and requiring additional manual curation to

maintain clarity. Another process that appeared to readily produce excessive information was tracking every variable that is used throughout a function that appears within the trace. While it is possible to track definitions and usage, without backpropagation, it is extremely likely that a giant list of variables will be generated that cannot be given useful concrete values. Behavior witnessed in our test cases suggest that while variables are defined in functions, there are also many variables that are references to external globals, set to macro calls that cannot consistently be tied to the file from the trace, or are user-defined at runtime. Tracking all of these variables and all of their possible values eventually became something that was out of scope for what could be accomplished, although the functionality for creating the initial variable defined and usage lists was left in the codebase it is not called by our current solution.

Discussion

Trace refinement enhances developer comprehension by carefully balancing completeness with readability. By reducing redundant or misleading information while adding contextual annotations, refined traces allow developers to follow the causal chain of a heap buffer overflow without excessive manual effort. This balance is critical, as overly verbose traces overwhelm users and lead to fatigue, while overly aggressive reduction risks omitting critical causal information.

These practical gains matter because refined traces integrate directly into existing analysis ecosystems. Symbolic execution tools can emit refined reports without altering their core detection logic. Machine-learning systems can rank or weight refined steps more effectively when the underlying structure is coherent. This flexibility

demonstrates that trace refinement is not only a post-processing improvement but can serve as a foundational layer for broader developer-facing or automated security analysis frameworks.

The work also exposes theoretical implications for how analysis tools should be composed. Layered approaches, for example, combining symbolic reasoning, source-level inspection, and structured RCA methods, produce richer, more stable traces than any technique on its own. This suggests a direction for future research: hybrid methodologies that integrate multiple partial views of how a program behaves to produce a single, clearer, and more complete explanation. This research shows that neither source inspection nor symbolic state alone is sufficient. The interaction between them is what yields a trace that developers can make use of.

These advantages sit alongside concrete challenges. Edge cases remain difficult to handle reliably. Functions with many variables or complex control flow patterns undermine automated reduction, and attempts to generalize the refinement rules across all traces could introduce bloat. Specialized analyses generate specialized artifacts, and widening their applicability requires more than deterministic rules. It requires deliberate design choices that trade strict generality for practical usability. Semi-automated analysis risks producing overlong traces if the underlying source structure is dense or irregular.

Altogether, the results show that refinement mitigates the limitations of specialized analyzers by converting sparse, repetitive output into clear, actionable artifacts. The approach improves interpretability without compromising precision, enabling developers to perform RCA more quickly and with higher confidence.

References

Guo, Yiyuan, et al. “Precise Compositional Buffer Overflow Detection via Heap Disjointness.” Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024), 2024, pp. 63-75. Association for Computing Machinery, https://doi.org/10.1145/3650212.3652110. Accessed 29 September 2025.

Li, Lian, et al. “Practical and effective symbolic analysis for buffer overflow detection.” Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE '10), 2010, pp. 317-326. Association for Computing Machinery, https://doi.org/10.1145/1882291.1882338. Accessed 29 September 2025.

OWASP. “Buffer Overflow”, https://owasp.org/www-community/vulnerabilities/Buffer_Overflow/. Accessed September 29, 2025.

OSS-Fuzz. Google, https://google.github.io/oss-fuzz/. Accessed 29 September. 2025.

Zhang, Chenxi, et al. “Trace-based Multi-Dimensional Root Cause Localization of Performance Issues in Microservice Systems.” Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), 2024, pp. 1-12. Association for Computing Machinery, https://doi.org/10.1145/3597503.3639088. Accessed 24 November 2025.