CV: Interactive Visualization of Verification Results

Vitalii Mordan*† ✉ , Vadim Mutilin*†‡
*Trusted AI Research Center, RAS, Moscow, Russia
†Ivannikov Institute for System Programming, RAS, Moscow, Russia
‡Moscow Institute of Physics and Technology (State University), Moscow Region, Russia

***Abstract*—We present CV (Continuous Verification), a framework for regression verification that transforms machine-readable, SV-COMP–compliant results into interactive, human-readable visualizations. CV assists software developers by presenting error traces, correctness proofs, and overall statistics in a concise and accessible form, thereby reducing the effort required for manual analysis of verification results. In addition, it supports regression verification by comparing outcomes across software versions and highlighting new results – whether potential bugs or correctness proofs. Although originally developed for SV-COMP, CV is applicable to any verification benchmark following the SV-COMP format, including real-world software such as Linux kernel modules. CV is available as open-source software. A video demonstration is available at https://youtu.be/98JzOwgKmjY.**

***Index Terms*—software verification, witness visualization, SV-COMP, verification witness, manual analysis, bug localization, correctness proof.**

I. Introduction

Software verification serves as a formal method for automatic examining a program's source code without executing it, traversing all possible program paths. The objective is to prove program correctness against specified properties under corresponding assumptions, in contrast to testing techniques and bug finding methods like static analysis. Modern software verification tools are capable of automatically analyzing large codebases and producing detailed results about potential bugs or correctness proofs.

In practice, verification results are typically produced in the standard result representation established by the Competition on Software Verification (SV-COMP) [1]. This format is supported by numerous state-of-the-art software verifiers and is well suited for automated processing. However, it is not designed to be human-readable, which complicates manual inspection.

Manual inspection of verification results remains an essential step in the software development workflow, particularly when identifying and confirming real bugs. Interpreting raw witness files is often time-consuming and error-prone, especially for large and complex programs. The situation becomes even more challenging when analyzing successive versions of a software system. For example, in Google Summer of Code (GSoC) projects focused on Linux driver analysis, only a small fraction of reported issues corresponded to real bugs, while the full review process required several months[1]. Given the frequency of new Linux kernel releases, it becomes evident why software verification methods are still rarely applied in continuous development practice.

To address these challenges, we introduce Continuous Verification (CV), a framework that turns machine-readable verification results into an interactive, human-readable format. CV takes results in the SV-COMP format together with the corresponding source code and presents them through a web interface that supports both visualization and regression analysis. It supports manual review of all detected potential bugs and correctness proofs, while in regression workflows it highlights only newly introduced results, helping developers focus on relevant changes.

Originally designed for SV-COMP witnesses, CV has also been applied to Linux kernel module verification, showing its generality for any results conforming to the SV-COMP format.

By making verification results more accessible and easier to interpret, CV reduces the time required for manual analysis and increases the practical usability of modern software verification tools.

This paper makes the following contributions:

- We present CV, a framework that transforms arbitrary SV-COMP–compliant verification results into interactive, human-readable visualizations.
- We extend CV with regression verification support by detecting new and unique results across program versions.
- We demonstrate the applicability of CV on multiple versions of Linux kernel benchmarks, illustrating its potential for industrial-scale verification.

II. Background

A. Definitions

Software verification aims to determine whether a program satisfies specified properties, such as memory safety or the unreachability of an error function. A **verification task** consists of the program source code, an entry point (e.g., the `main()` function), and the properties to be checked. A **benchmark** is a collection of verification tasks, typically grouped by a common property or another shared parameter.

A **software verifier** is a tool that processes a verification task and produces a **verification result**, which falls into one of the following categories:

- TRUE: the property holds for all executions; a correctness proof may be provided. If the proof is incorrect, this situation is labeled as a **missed bug**.

✉ Email: mordan@ispras.ru

[1]See, for instance, http://linuxtesting.org/20-08-2021, http://linuxtesting.org/31-08-2020, and http://linuxtesting.org/28-08-2020.

- FALSE: a property violation was found, and one or more error traces are returned. If a particular error trace does not correspond to a real bug in the program, it is considered a **false alarm**.
- UNKNOWN: the verifier could not complete the task (e.g., due to a timeout, resource exhaustion, or an internal error).

A **witness** is a machine-readable artifact that accompanies a verification result, describing either a correctness proof or an error trace. An **error trace** consists of one or more paths of program operations that lead from an entry point to a property violation. A **correctness proof**, on the other hand, is a graph that covers all potential executions of the program from the entry point, demonstrating the absence of any property violation [2]. In this work, we focus on witnesses that conform to the GraphML-based format standardized by SV-COMP [3].

Witnesses are structured as directed graphs, where nodes and edges encode control flow and semantic information. Edges in the witness graph correspond to program operations, each linked to the relevant line of source code. We classify these operations into the following categories:

- **Function calls** represent entering and returning from functions, forming the structural backbone of an error trace.
- **Conditions** indicate which branch of a conditional statement was taken during execution.
- **Assumptions** specify values assumed to hold for variables, function arguments, or return values.
- **Thread information** describe thread identifiers and thread creation events, essential for analyzing concurrent programs.
- **Loop heads** mark the entry points of loops, constraining possible transitions in the program's control flow.
- **Invariants** specify conditions expressed as C formulas that must hold within a given scope, used only in correctness proofs.

If we intend to apply software verification in a real-world scenario, the first step is to create a benchmark, i.e., to generate verification tasks for the given source code and selected properties, and then solve them using any SV-COMP–compliant verifier. Afterwards, manual analysis becomes essential. Error traces need to be inspected and categorized into false alarms and real bugs, with the latter requiring attention from software developers. When the objective is to prove correctness, the challenge becomes even greater: proofs must be carefully examined to identify potential missed bugs. Due to their inherent complexity, such proofs are difficult to analyze and require substantial expertise. This paper focuses on presenting verification results in a human-readable form, thereby reducing the effort of manual inspection – a step that remains a major bottleneck in the verification workflow.

B. Related Work

BENCHEXEC [4] is the benchmarking framework used in SV-COMP to execute software verifiers and collect standardized outputs. It produces results in the SV-COMP format and works with any SV-COMP verifier. In addition to witness files, it generates detailed tables, statistics, and visual summaries of resource usage and tool performance, which are invaluable for fair and reproducible competition-style evaluations. However, BENCHEXEC is primarily designed for comparing verification tools on predefined benchmarks rather than for presenting results to software developers. It lacks human-oriented visualization and does not facilitate the manual analysis of potential bugs or correctness proofs.

Some verifiers, such as CPACHECKER [5], provide internal visualizations of results, but these rely on tool-specific formats and are not universally accessible.

SV-COMP does not require participants to provide visualizers, leaving this task to verifier developers or third-party tools. One such tool is KLEVER [6], which generates tasks from large-scale projects (e.g., Linux kernel, BUSYBOX), executes them with CPACHECKER, and visualizes error traces. Its interface highlights program structure through calls, returns, branches, thread contexts, and coverage information, offering developers more accessible feedback than raw witnesses. However, KLEVER is tightly bound to its own back-end and formats, does not support arbitrary SV-COMP tools, provides limited capabilities for regression verification, and does not visualize correctness proofs.

The Static Analysis Results Interchange Format (SARIF) [7] provides a standardized representation of static analysis results, with built-in support for regression tracking across program versions. While SARIF is designed primarily for bug-finding and does not support verification witnesses – making it unsuitable for visualizing correctness proofs or structured error traces – its mechanisms for highlighting new or changed issues across versions are highly relevant. This motivates to create a similar approach in software verification.

Visualization has also been explored in adjacent domains. For example, PMC-VIS [8] visualizes Markov chains in probabilistic model checking, allowing users to explore probabilistic transitions and analyze quantitative properties over states. Similarly, [9] introduces a framework for visualizing hyperproperties by graphically representing relationships across multiple execution traces. While both approaches demonstrate how complex internal graph structures can be effectively visualized, their representations are domain-specific: Markov chains and hyperproperty graphs differ from verification witnesses. As a result, these techniques cannot be directly reused for software verification results. Nevertheless, they highlight an important point: even highly complex and domain-specific graph structures can be visualized, motivating our effort to provide effective visualizations for verification witnesses.

Thus, no existing tool combines visualization, interpretation, and regression support for SV-COMP results – the gap that CV aims to fill.

III. HUMAN-READABLE VISUALIZATION OF VERIFICATION RESULTS

We present CV (Continuous Verification), a framework that transforms SV-COMP–compliant verifier outputs into human-

readable feedback. The framework provides a bridge between raw BENCHEXEC benchmarking output and human-readable feedback, enabling developers and researchers to analyze verification outcomes more effectively.

The architecture consists of two components:

1) A preprocessing layer[2] that converts BENCHEXEC result directories and source code into an internal format.
2) A web-based visualization layer[3] that renders results and enables interactive exploration, including regression comparisons.

CV supports two modes: (i) standalone, for inspecting a single verification result such as an error trace, and (ii) comparative, for analyzing complete benchmark results in the web interface.

A. Visualization of Error Traces

For verification outcomes classified as FALSE, CV provides structured visualization of the corresponding error traces. The design is inspired by Klever-style techniques [6], but generalized to support arbitrary SV-COMP–compliant verifiers and benchmarks. Each trace is rendered in a clear, hierarchical format that emphasizes semantically relevant operations while omitting unnecessary detail.

The following witness elements are visualized (each operation links directly to the corresponding source code line):

- **Function calls** are displayed as a hierarchical call tree. Each invocation opens a block that closes upon return, forming the backbone of the trace.
- **Conditions** appear as C-expressions evaluated to `true`.
- **Assumptions** represent evaluated expressions (e.g., `<VAR> == <VALUE>`) that describe the program state.
- **Thread information** is indicated with distinct visual markers, clarifying which thread is active.
- **Loop heads** denote locations where execution repeatedly cycles.
- **Violation hints** mark operations directly related to a property violation and serve as visual anchors for understanding the cause of the bug.

When witness elements are missing or incomplete, CV reports this to the user, providing feedback that helps verifier developers improve the visualization quality.

As shown in [10], witness quality can be assessed by the presence of *relevant elements* – those that enhance automatic witness processing and facilitate human comprehension. However, relevant elements strongly depend on both the verified property and the benchmark set. Even witnesses containing the same element types (e.g., assumptions) may differ significantly in diagnostic value. To address this limitation, we introduce a new, property-independent concept of *violation hints*.

Violation hints extend the SV-COMP witness format with explicit indicators of operations most relevant to a property violation (e.g., invalid pointer dereferences, data races, or incorrect clock usage). Depending on the property, the verifier can attach these hints to any existing element type, making them a lightweight yet effective mechanism to guide user attention. By default, CV collapses all parts of the call hierarchy that contain no violation hints and hides non-essential elements, allowing users to expand details on demand.

Currently, only a subset of verifiers (e.g., CPACHECKER [5]) supports the generation of violation hints, giving them a clear advantage in visualization quality. Nevertheless, CV remains fully compatible with the SV-COMP witness standard and can process and display witnesses both with and without these annotations.

B. Visualization of Correctness Proofs

While error traces have long been visualized, correctness proofs remain largely unexplored from a usability standpoint. A correctness proof, typically represented as a graph (e.g., an abstract reachability graph [2]), can be rechecked automatically but is often enormous and unintuitive for developers. Moreover, many SV-COMP verifiers still produce empty proofs (see the artifact [11]), which further limits their practical value. As a result, correctness proofs are rarely inspected in real workflows.

Yet, understanding why a bug was not reported can be equally important — especially when a verification result may be unsound or incomplete. To support such cases, CV introduces an initial, developer-oriented view of correctness proofs that focuses on interpretable elements rather than raw proof structures.

a) Source Code Coverage: CV highlights the parts of source code actually explored during verification and indicates under which conditions branches were taken or pruned. For each line, the explored conditions derived from the proof are unified into a simplified logical expression, helping developers identify unverified regions where potential bugs may still remain. Since proofs may include a large number of conditions, the interface allows users to search for specific variables of interest, thereby filtering the displayed conditions. This functionality is particularly useful when rechecking recently modified parts of the program – for example, to verify that a previously fixed bug is now correctly handled.

b) Invariants: All invariants discovered within a given scope are aggregated and displayed in disjunctive form. This enables reasoning about loop termination and bounded variable ranges, clarifying why certain behaviors were classified as safe and helping identify potential weaknesses in the analysis.

While CV currently visualizes only the available proof elements, we envision extending this approach to also highlight potential blind spots in the proof – for instance, regions of code or behaviors not covered due to verifier assumptions or configuration constraints. For example, some verifiers assume that memory allocations never fail – an optimization that reduces false alarms but may also conceal real bugs. Such functionality would help users identify where a proof might fail to detect errors. Implementing this feature would require extending the SV-COMP format to explicitly represent verifier assumptions and heuristics.

[2]CV public repository: https://github.com/ispras/cv.

[3]Web interface public repository: https://github.com/ispras/cv-visualizer.

In summary, CV deliberately focuses correctness proof visualization on interpretable and relevant aspects, marking an initial step toward practical, comprehensible, and developer-oriented proof exploration.

C. Visualization of Benchmarks Verification Results

When a verification run consists of many tasks, CV unifies them into a single benchmark-level view. This design follows the spirit of BENCHEXEC tables but is tailored for interactive exploration rather than competition scoring. The aggregated view provides a quick overview of verification outcomes across an entire benchmark suite or software system.

For each benchmark, CV presents all collected artifacts: error traces, correctness proofs, and verifier logs for UNKNOWN cases. These are complemented with detailed resource statistics (CPU time, wall time, memory and disk usage), including quantile plots that help evaluate scalability. For larger systems, aggregated code coverage highlights which parts of the software were analyzed. If available, verifier-specific metadata is also integrated into the overview.

Thus, CV is designed to transform raw benchmark outputs into accessible visualizations that help developers understand all the verification outcomes. Examples of the visualized results for benchmarks are available in the artifact [11].

IV. REGRESSION VERIFICATION SUPPORT

Modern software systems evolve continuously, with frequent updates and feature extensions. In such environments, regression verification is essential to ensure that previously fixed bugs do not reappear and that newly introduced errors are promptly detected. To support this process, CV provides dedicated mechanisms for comparing verification results across program versions and highlighting differences between them.

To demonstrate these capabilities, we evaluated CV on verification tasks derived from Linux kernel modules (*usb* and *net* drivers) across three consecutive versions (from October 17, 2024 to March 13, 2025). The tasks were prepared with KLEVER for three representative properties and solved by CPACHECKER via BENCHEXEC. An accompanying artifact [11] includes all benchmarks, preprocessed visualized results with docker image, installation instructions, and comparison tables for witnesses produced by various verification tools. Table I summarizes the verification outcomes processed by CV.

A. Filtering of Equivalent Error Traces

For each FALSE result, a software verifier may generate multiple error traces, since a property can often be violated in several distinct ways. In practice, many of these traces correspond to the same underlying bug [12], differing only in minor syntactic or structural details. From a user perspective, analyzing more than one such trace is redundant. Moreover, even when a program contains only a single real bug, regression verification requires comparing traces across versions to determine whether it is already known or newly introduced.

To address this, CV applies *Equivalent Trace Filtering* (ETF) [12] by default during both error trace processing and regression analysis. ETF groups similar error traces into equivalence classes based on their internal structure, where each class represents a distinct potential bug. The user then inspects only one representative trace (typically the first produced by the verifier), which is sufficient to validate the bug or mark it as a false alarm. Equivalence can be determined using multiple criteria, such as matching function-call hierarchies with violation hints at the leaves (the default), or comparing assumptions, conditions, and other witness elements.

As shown in Table I, raw verification often produces thousands of error traces (column "Raw traces"), while CV uses ETF to collapse them into a few hundred unique ones ("Unique traces"), making manual inspection feasible and ensuring that all violations are reviewed rather than only the first. This effect is most evident for the *memory safety* property, which produces a large number of similar traces caused by common pointer-related issues in C. Without ETF, manual inspection of such results would be impractical.

Despite processing thousands of traces, CV demonstrates low computational overhead: its own runtime ("CV time (s)") remains negligible compared to the CPACHECKER total CPU time ("Verifier time (s)"), confirming that filtering scales efficiently even for complex verification tasks.

B. Comparison of Verification Results Across Versions

To assess regression verification, CV compares verification outcomes between two benchmark sets, typically corresponding to different versions of the same system. The primary goal is to identify *new results* — either newly discovered error traces or additional correctness proofs — which indicate behavioral differences introduced by recent code changes. Such results are automatically highlighted, allowing developers to focus on the most relevant regressions and improvements while ignoring unchanged outcomes. In addition, *absent results* can be equally informative, for example, corresponding to fixed bugs.

Beyond new and absent results, CV also compares aggregate verification statistics, including resource usage (e.g., CPU time and memory consumption, visualized through quantile and scatter plots), code coverage, and total counts of verification outcomes per property. Together, these metrics provide a concise, data-driven overview of how verification behavior evolves across versions.

As shown in Table I, only a small number of new results appear between kernel versions, demonstrating that CV effectively narrows the developer's focus to truly changed behaviors — a perspective not offered by BENCHEXEC or KLEVER. The property-wise comparison also reveals distinct verification dynamics: the gradual reduction in trace counts for *memory safety* suggests ongoing improvements in kernel robustness; the moderate churn observed in *concurrency safety* reflects the continuous appearance and resolution of data races; and the stable behavior of the *clock API* property aligns

TABLE I: Verification results across Linux kernel versions and properties. In regression verification, CV detects new and absent results, reduces large trace volumes through ETF, and achieves this with negligible runtime cost.

Kernel	Proofs	Raw traces	Unique traces	New traces	New proofs	Absent traces	Absent proofs	Verifier time (s)	CV time (s)
Memory safety property (*MemSafety* in SV-COMP): detects invalid pointer dereferences, invalid memory deallocations, and memory leaks									
5.10.227	458	28 096	952	–	–	–	–	342 000	13 400
5.10.231	453	28 155	946	+1	+1	-7	-6	343 000	13 300
5.10.235	453	28 693	957	+12	–	-1	–	342 000	13 300
Concurrency safety property (*ConcurrencySafety* in SV-COMP): checks for data races in multi-threaded code									
5.10.227	631	59	45	–	–	–	–	175 000	370
5.10.231	631	62	47	+8	–	-6	–	174 000	390
5.10.235	629	52	37	+4	–	-14	–	175 000	370
Clock API property (*ReachSafety* in SV-COMP): checks correct usage of Linux kernel clock primitives									
5.10.227	1 547	36	36	–	–	–	–	161 000	2 250
5.10.231	1 549	33	33	–	+6	**-3**	-4	160 000	1 770
5.10.235	1 549	31	31	–	+3	**-2**	-3	162 000	2 210

with its smaller and more specialized role within the kernel infrastructure.

C. Persistent User Marks Across Versions

To further reduce manual effort, CV supports interactive marking of verification results. Once a developer classifies an error trace (e.g., as a real bug or a false alarm), this annotation is stored and automatically propagated to all previous and subsequent program versions where the same bug persists. Users can refine these marks as needed – for example, by indicating which parts of a trace are essential for comparison via ETF.

Newly discovered bugs appear as unmarked items, making them immediately visible as potential regressions. This persistence mechanism enables developers to maintain a consistent knowledge base of verification outcomes throughout the software's evolution, avoiding redundant manual reviews.

The practical impact of this feature is illustrated by the *clock API* property: manual inspection with CV identified 17 true positives (47% of reported traces), five of which were later confirmed and fixed upstream – directly reducing the number of reported bugs (see Table I, column "Absent traces" for the *clock API* property).

V. Conclusion

We presented CV, a framework for continuous regression verification that transforms SV-COMP–compliant benchmark results into human-readable visualizations. The tool supports detailed inspection of error traces and correctness proofs and provides aggregated statistics to facilitate a comprehensive understanding of verification outcomes. CV is available as open-source software, with installation instructions and complete examples provided in the accompanying artifact.

CV extends traditional verification workflows by enabling automated comparison across software versions, highlighting newly introduced results, and applying advanced error trace filtering to reduce manual effort.

Our experiments on benchmarks derived from several Linux kernel versions demonstrate the practical applicability of CV in real-world regression analysis, illustrating its potential to improve both efficiency and interpretability in large-scale software verification.

References

[1] D. Beyer and J. Strejček, "Improvements in software verification and witness validation: SV-COMP 2025," in *Proc. TACAS*, A. Gurfinkel and M. Heule, Eds. Springer Nature Switzerland, 2025, pp. 151–186. [Online]. Available: https://doi.org/10.1007/978-3-031-90660-2_9

[2] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann, "Correctness witnesses: exchanging verification results between verifiers," in *Proceedings of the 2016 24th ACM SIGSOFT*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 326–337. [Online]. Available: https://doi.org/10.1145/2950290.2950351

[3] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig, "Verification witnesses," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, Sep. 2022. [Online]. Available: https://doi.org/10.1145/3477579

[4] D. Beyer, "Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016)," in *Proc. TACAS*, ser. LNCS 9636. Springer, 2016, pp. 887–904. [Online]. Available: https://doi.org/10.1007/978-3-662-49674-9_55

[5] D. Beyer and M. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Proc. CAV*, ser. LNCS. Springer Berlin Heidelberg, 2011, vol. 6806, pp. 184–190. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_16

[6] E. Novikov and I. Zakharov, "Verification of operating system monolithic kernels without extensions," in *Proc. ISoLA*. Berlin, Heidelberg: Springer-Verlag, 2018, p. 230–248. [Online]. Available: https://doi.org/10.1007/978-3-030-03427-6_19

[7] M. C. Fanning and L. J. Golding, "Static analysis results interchange format (SARIF) version 2.1.0," OASIS Committee Specification, March 2020. [Online]. Available: https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html

[8] M. Korn, J. Méndez, S. Klüppelholz, R. Langner, C. Baier, and R. Dachselt, "PMC-VIS: An interactive visualization tool for probabilistic model checking," in *Software Engineering and Formal Methods: 21st International Conference*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 361–375. [Online]. Available: https://doi.org/10.1007/978-3-031-47115-5_20

[9] T. Horak, N. Coenen, N. Metzger, C. Hahn, T. Flemisch, J. Méndez, D. Dimov, B. Finkbeiner, and R. Dachselt, "Visual analysis of hyperproperties for understanding model checking results," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 1, p. 357–367, Jan. 2022. [Online]. Available: https://doi.org/10.1109/TVCG.2021.3114866

[10] V. Mordan and V. Mutilin, "Software verifcation witnesses thoroughness," *Programming and Computer Software (to appear)*, vol. 52, no. 1, 2026.

[11] V. Mordan, "Artifact: Visualization of verification results for Linux kernel benchmarks with CV," Nov. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17371298

[12] V. Mordan and E. Novikov, "Minimizing the number of static verifier traces to reduce time for finding bugs in Linux kernel modules," in *Proc. SYRCoSE*, no. 8, 2014. [Online]. Available: https://doi.org/10.15514/SYRCOSE-2014-8-5