Comparing Transparent Static Analyzers with Open Verification Dashboard

Tom Goalard[1]
ENS Rennes, Campus de Ker Lann, 11 avenue Robert Schuman, 35170 Bruz, France

Karoliine Holter[1]
University of Tartu, Institute of Computer Science, Narva mnt 18, 51009 Tartu, Estonia

Simmo Saan
University of Tartu, Institute of Computer Science, Narva mnt 18, 51009 Tartu, Estonia

Vesal Vojdani
University of Tartu, Institute of Computer Science, Narva mnt 18, 51009 Tartu, Estonia

Raphaël Monat
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, 59000 Lille, France

Abstract

Given an input program, sound static analyzers compute a list of potential runtime errors in it. However, measuring their precision and comparing their results remains challenging. In this work, we formalize a notion of *transparent static analyzers* that report the proof obligations they check, including both verified and unverified obligations. This transparent output enables a semantics-directed, fine-grained comparison and the combination of static analyzers. We introduce the Open Verification Dashboard (OVD), which provides a unified interface to aggregate the results of multiple static analyzers. By juxtaposing verified properties and outstanding warnings, OVD highlights coverage gaps, variabilities and inconsistencies across tools. We experimentally evaluate the benefits of OVD on benchmarks from the Competition on Software Verification (SV-COMP). This work paves the way for a static analysis standard for C runtime error reporting.

2012 ACM Subject Classification Software and its engineering → Automated static analysis; Software and its engineering → Software testing and debugging

Keywords and phrases automated static analysis, multi-tool integration, interoperability, proof obligations, result aggregation, verification progress, selectivity metric, reproducibility, dashboard

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.

Supplementary Material *Artifact*: `https://doi.org/10.5281/zenodo.19651018`
Software (Source Code): `https://github.com/sws-lab/open-verification-dashboard`
archived at `swh:1:dir:f36521d14e7e5034a524502244b9240f7685e760`

Funding This work is supported by the European Union and the Estonian Research Council via projects PRG2764 and TEM-TA119, by grant agreement ANR-24-CE25-7956-01 RAISIN from the French Agence Nationale de la Recherche, and by an Amazon Research Award, Fall 2024.

1 Introduction

Sound static analyzers, such as Frama-C's Eva plugin [9], Goblint [48, 43] and Mopsa [25, 36], can be used to *prove the absence* of runtime errors in C programs. In practice, sound analyzers perform over-approximations for their analyses to terminate, and report a list of *potential* runtime errors, called *alarms*. Once this report is available, a key difficulty faced by static analysis users is to discriminate between *true* and *false alarms*.

[1] Equal contribution.

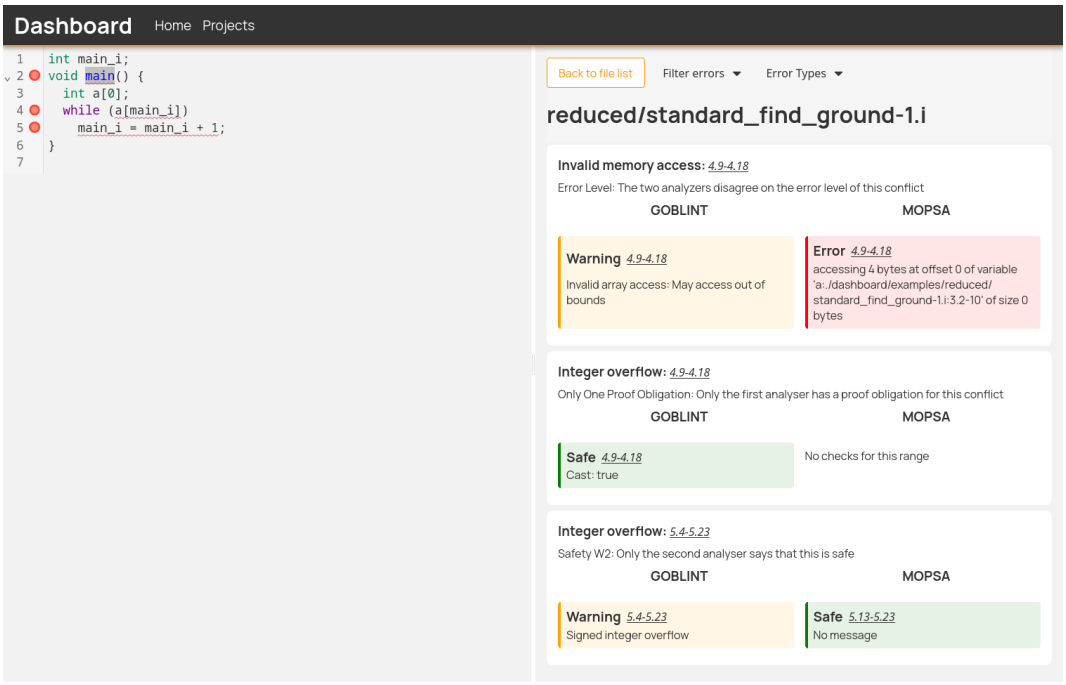


Figure 1 Screenshot of the dashboard GUI. The left panel shows the code with report differences highlighted, the right panel shows the POs and their statuses.

Alarms have been one of the main targets of user interface and user experience research around static analyzers. For example, they have been identified as one of the key factors dissuading developers from using static analyzers [13]. As of 2023, a survey identified 130 primary research articles focusing on alarm post-processing [39]. However, alarms reported by analyzers tell only one side of the story: a user seeing no alarm from a tool cannot easily tell whether the tool actually checked a given property or simply did not consider it, e.g., due to unsound behavior.

Deductive verification frameworks such as Frama-C's WP plugin [15] and Why3 [19] keep track of *proof obligations (POs)* that have to be discharged by automated or interactive theorem provers [28]. There, verification condition generation and discharge are decoupled, so one can export the status of each PO to a file or display it in an IDE. On this analogy, every safety check performed by a static analyzer can be viewed as a PO, i.e., a proposition that must be proved *valid* or reported *invalid/unknown*. Mopsa has recently added the capability to perform *transparent reporting*: it displays all checks performed, both failed and successful, in an effort to compute automated precision metrics [37].

Even with some transparent reporting, comparing the reports of different static analyzers remains a challenge. On the theoretical side, different analyzers may rely on incomparable over-approximations. On the practical side, it is difficult to transform these tool-specific PO reports into a unified view of what each tool verified, under which assumptions, and where their results concur or conflict. As static analyzers tightly couple the generation of POs with their discharge, the resulting PO reports, which are reconstructed by tracking the checks performed, will depend on how program expressions are decomposed and how the tool categorizes the checks. For example, if there is an out-of-bounds access when copying a string using `strcpy`, one tool may categorize this as a "buffer overflow", while another may consider it a violation of the `strcpy` contract. One core difficulty therefore lies in *harmonizing* these

heterogeneous PO reports.

In this work, we study the benefits of transparent reporting from static analyzers and experimentally compare analysis results using a comparative verification dashboard. The resulting comparisons led to mutual improvements in the C analyses of Goblint and Mopsa. We show an example of the comparisons enabled by the dashboard in Figure 1. The program in the left panel is the result of an automated test-case reduction [41] on a program from the Competition on Software Verification (SV-COMP) [7]. The right panel shows POs where Goblint and Mopsa disagree. In the first case, both analyzers report an alarm regarding the array access, but report different levels of severity. The second case shows that Goblint encounters an integer cast PO that Mopsa does not, hinting at a potential soundness issue in Mopsa or a superfluous PO generated by Goblint. In the third case, Mopsa is able to discharge the PO, while Goblint raises an alarm, which can be seen as a precision issue. Note that Goblint and Mopsa do not report the same program locations in this last PO, but the dashboard has been able to align them. The workflow enabled by this work paves the way for a standardized classification for C runtime error reporting. We created a preliminary classification for alarms related to integer overflows, which we implemented inside both Goblint and Mopsa. In the future, we envision using dashboard reports in long-term continuous-integration static analysis pipelines, so that users can focus their analyses on less-covered code or on cases where the analysis does not reach satisfactory precision.

Contributions. This work presents the following contributions:

- We provide a novel formal framework for transparent static analysis (TSA) in Section 2. We illustrate the framework through a concrete semantics on a toy imperative language, a non-relational static analysis to detect arithmetic errors, and a soundness theorem.
- We describe a classification of comparison outcomes for transparent static analyzers in Section 3 and define joint analyzer combinations with fine-grained collaboration between them.
- We implemented the Open Verification Dashboard (OVD), providing comparison methods for generic transparent static analyzers. OVD is presented in Section 4.
- We experimentally evaluate the benefits of TSA and OVD in Section 5, on C programs from SV-COMP focusing on integer overflows. We compare two state-of-the-art analyzers participating in SV-COMP: Mopsa, and a version of Goblint extended to make it a transparent static analyzer. Throughout the comparison process, we identified and helped fix 14 bugs in Goblint and Mopsa.

2 A Formal Framework for Transparent Static Analysis

This section introduces transparent static analysis (TSA), a novel framework where both the concrete semantics and the analyzer collect the proof obligations related to runtime error detection. We illustrate it on a toy imperative language with integer overflows and divisions by zero, and provide updated soundness criteria relating the semantics and the analysis.

Program verification is the task of proving that a program satisfies a given specification. For flexibility and performance reasons, most practical programming languages provide unsafe language features, such as pointer arithmetic or unchecked array access. Hence, we want to verify that the use of such features does not lead to runtime errors, such as buffer overflows or null pointer dereferences. The specification is thus the absence of runtime errors anywhere in the program, so the static analyzer needs to discover all potential verification conditions that must hold to ensure that the program is safe.

$$
\begin{array}{rll}
\mathsf{Expr} ::= & v & v \in \mathsf{Var} \\
\mid & m & m \in \mathbb{M} \\
\mid & \texttt{rand}(m_1, m_2) & m_i \in \mathbb{M}, m_1 < m_2 \\
\mid & \mathsf{Expr} \oplus^{\ell} \mathsf{Expr} & \oplus \in \{+, -, \times, /\}, \ell \in \mathcal{L} \\
\mathsf{Stmt} ::= & v = \mathsf{Expr} & v \in \mathsf{Var} \\
\mid & \texttt{if } \mathsf{Expr} \bowtie 0 \texttt{ then } \mathsf{Stmt} \texttt{ else } \mathsf{Stmt} \texttt{ endif} & \\
\mid & \texttt{while } \mathsf{Expr} \bowtie 0 \texttt{ do } \mathsf{Stmt} \texttt{ done} & \bowtie \in \{\leq, <, >, \geq, =, \neq\} \\
\mid & \mathsf{Stmt} \texttt{ ; } \mathsf{Stmt} &
\end{array}
$$

Figure 2 Syntax of the considered toy imperative language.

In program verification, a proof obligation is a logical statement that needs to be proven to ensure that the program satisfies the specification. For the purpose of proving the absence of runtime errors, we consider each potential runtime error site as a PO. More precisely, for each potentially unsafe expression occurrence in the program and each relevant safety category, we introduce one PO. We write $\mathcal{L}$ for the opaque set of locations in the program and $\mathcal{C}$ for the finite set of abstract safety check categories (e.g., invalidMemAccess, intOverflow, divByZero, ...). $\mathcal{C}$ should be general enough to compare analyzers with different category granularities. Hence, if an analyzer can report a finer-grained or different category, it should map it to one of the existing categories. Formally, each PO is the pair $\varphi = (\ell, c) \in \mathcal{L} \times \mathcal{C}$ which we interpret as the statement "c does not occur at *location* ℓ".

Syntax. Let Var be the set of program variables. Consider a fairly standard imperative toy programming language using the syntax in Figure 2, operating on a finite set of machine integers $\mathbb{M} \subseteq \mathbb{Z}$, e.g., 64-bit signed integers. The `rand` expressions can be used to model nondeterminism and the environment. The binary operation expressions are labelled with their locations $\ell \in \mathcal{L}$. For simplicity, binary operations will be the only source of runtime errors in the toy language; thus, locations for other expressions and statements will not be necessary for the example.

Concrete Collecting Semantics. Since our toy language operates on machine integers, runtime errors can be due to integer overflows and divisions by zero. Consider the concrete semantics partially shown in Figure 3. The evaluation of an expression e in an environment σ yields a set of all possible values of e. Contrary to textbook definitions of static analyses [35, 33] which drop erroneous states for the sake of simplicity, our concrete semantics also collects the set of all safety checks performed to ensure the absence of runtime errors. Note that the returned set of checks may contain both $(\varphi, \checkmark)$ and $(\varphi, \text{✗})$ for the same proof obligation φ, if the corresponding operation may succeed or fail on different operand values. The checks for runtime errors are collected recursively from subexpressions, while the values come only from error-free evaluations. The execution of a statement s in a set of environments S yields a set of all possible environments after executing s and a set of all safety checks performed. For brevity, we only spell out the semantics for some kinds of expressions and statements, the rest is similar and standard. It is vital that the returned checks and non-failing values are computed and returned separately, so that if the set of values ends up being empty due to a definite runtime error, then the failing check must still be returned.

$\mathbb{M} \subseteq \mathbb{Z}$	*Machine integers*
$\sigma \in \Sigma = \mathsf{Var} \to \mathbb{M}$	*Concrete environments*
$\ell \in \mathcal{L} \subseteq \mathbb{N}$	*Program locations*
$\mathcal{C} = \{\mathsf{intOverflow}, \mathsf{divByZero}\}$	*Safety check categories*
$\varphi \in \mathsf{PO} = \mathcal{L} \times \mathcal{C}$	*Proof obligations*
$\Theta \subseteq \mathsf{CH} = \mathsf{PO} \times \{\checkmark, \text{✗}\}$	*Checks*
$\mathbb{E}[\![e \in \mathsf{Expr}]\!] : \Sigma \to 2^{\mathbb{M}} \times 2^{\mathsf{CH}}$	*Concrete semantics of* e
$\mathbb{S}[\![s \in \mathsf{Stmt}]\!] : 2^{\Sigma} \to 2^{\Sigma} \times 2^{\mathsf{CH}}$	*Concrete semantics of* s

For example:

$$\mathbb{E}[\![e_1 +^{\ell} e_2]\!](\sigma) = \textbf{let } (M_1, \Theta_1) = \mathbb{E}[\![e_1]\!](\sigma) \textbf{ and } (M_2, \Theta_2) = \mathbb{E}[\![e_2]\!](\sigma) \textbf{ in} \quad \text{(add)}$$
$$\textbf{let } \Theta' = \{((\ell, \mathsf{intOverflow}), \mathrm{check}(m_1 + m_2 \in \mathbb{M})) \mid m_i \in M_i\} \textbf{ in}$$
$$(\{m_1 + m_2 \mid m_i \in M_i, m_1 + m_2 \in \mathbb{M}\}, \Theta_1 \cup \Theta_2 \cup \Theta')$$

$$\mathbb{E}[\![e_1 /^{\ell} e_2]\!](\sigma) = \textbf{let } (M_1, \Theta_1) = \mathbb{E}[\![e_1]\!](\sigma) \textbf{ and } (M_2, \Theta_2) = \mathbb{E}[\![e_2]\!](\sigma) \textbf{ in} \quad \text{(div)}$$
$$\textbf{let } \Theta' = \{((\ell, \mathsf{divByZero}), \mathrm{check}(m_2 \neq 0)) \mid m_2 \in M_2\} \textbf{ in}$$
$$\textbf{let } \Theta'' = \{((\ell, \mathsf{intOverflow}), \mathrm{check}(m_1 \,/\, m_2 \in \mathbb{M})) \mid m_i \in M_i, m_2 \neq 0\} \textbf{ in}$$
$$(\{m_1 \,/\, m_2 \mid m_i \in M_i, m_2 \neq 0, m_1 \,/\, m_2 \in \mathbb{M}\}, \Theta_1 \cup \Theta_2 \cup \Theta' \cup \Theta'')$$

$$\mathbb{S}[\![v = e]\!](S) = \textbf{let } S' = \{\sigma[v \mapsto m] \mid \sigma \in S, \mathbb{E}[\![e]\!](\sigma) = (M, \_), m \in M\} \textbf{ in} \quad \text{(assign)}$$
$$\textbf{let } \Theta' = \bigcup\{\Theta \mid \sigma \in S, \mathbb{E}[\![e]\!](\sigma) = (\_, \Theta)\} \textbf{ in}$$
$$(S', \Theta')$$

where

$$\mathrm{check}(b) = \textbf{if } b \textbf{ then } \checkmark \textbf{ else } \text{✗}$$

Figure 3 Concrete collecting semantics.

▶ **Example 1.** To illustrate the concrete semantics, consider the expression

$$e = \texttt{rand}(0, 1) +^{\ell_{\mathsf{ex}}} 1$$

at some location $\ell_{\mathsf{ex}} \in \mathcal{L}$. Assume $\texttt{rand}(m_1, m_2)$ evaluates to a machine integer in the interval $[m_1, m_2]$. For readability, let us instantiate the machine integers to the tiny range $\mathbb{M} = \{-1, 0, 1\}$. Thus, $2 \notin \mathbb{M}$ and adding $1 + 1$ triggers an overflow. Hence, for any environment $\sigma \in \Sigma$,

$$\mathbb{E}[\![\texttt{rand}(0, 1)]\!](\sigma) = (\{0, 1\}, \emptyset), \qquad \mathbb{E}[\![1]\!](\sigma) = (\{1\}, \emptyset).$$

Applying the addition rule (add) from Figure 3 gives

$$\Theta' = \{((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), \mathrm{check}(m_1 + m_2 \in \mathbb{M})) \mid m_1 \in \{0, 1\},\ m_2 \in \{1\}\}$$
$$= \{((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), \checkmark),\ ((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), \text{✗})\},$$
$$M' = \{m_1 + m_2 \mid m_1 \in \{0, 1\},\ m_2 \in \{1\},\ m_1 + m_2 \in \mathbb{M}\} = \{1\}.$$

Since $0 + 1$ does not overflow, whereas $1 + 1$ does, we return both a successful and a failing check; however, only the non-failing value 1 is included in the returned value set:

$$\mathbb{E}[\![e]\!](\sigma) = \Big(\{1\}, \{((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), \checkmark),\ ((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), \text{✗})\}\Big).$$

$\mathbb{Z}^\sharp$	*Abstract integers*
$\mathbb{M}^\sharp \subseteq^\sharp \mathbb{Z}^\sharp$	*Abstract machine integers*
$\Sigma^\sharp = \mathsf{Var} \to \mathbb{M}^\sharp$	*Abstract environments*
$\mathbb{E}^\sharp[\![e \in \mathsf{Expr}]\!] : \Sigma^\sharp \to \mathbb{M}^\sharp \times 2^{\mathsf{CH}}$	*Abstract semantics of e*
$\mathbb{S}^\sharp[\![s \in \mathsf{Stmt}]\!] : \Sigma^\sharp \to \Sigma^\sharp \times 2^{\mathsf{CH}}$	*Abstract semantics of s*

For example:

$$\begin{aligned}
\mathbb{E}^\sharp[\![e_1 +^\ell e_2]\!](\sigma^\sharp) = \textbf{let } (z_1^\sharp, \Theta_1) = \mathbb{E}^\sharp[\![e_1]\!](\sigma^\sharp) \textbf{ and } (z_2^\sharp, \Theta_2) = \mathbb{E}^\sharp[\![e_2]\!](\sigma^\sharp) \textbf{ in} \\
\textbf{let } \Theta' = \{((\ell, \mathsf{intOverflow}), \text{✗}) \mid z_1^\sharp +^\sharp z_2^\sharp \not\subseteq^\sharp \mathbb{M}^\sharp\} \textbf{ in} \\
\textbf{let } \Theta'' = \{((\ell, \mathsf{intOverflow}), \text{✓}) \mid (z_1^\sharp +^\sharp z_2^\sharp) \cap^\sharp \mathbb{M}^\sharp \neq \bot^\sharp\} \textbf{ in} \\
((z_1^\sharp +^\sharp z_2^\sharp) \cap^\sharp \mathbb{M}^\sharp, \Theta_1 \cup \Theta_2 \cup \Theta' \cup \Theta'') \\
\mathbb{S}^\sharp[\![v = e]\!](\sigma^\sharp) = \textbf{let } (z^\sharp, \Theta) = \mathbb{E}^\sharp[\![e]\!](\sigma^\sharp) \textbf{ in } (\sigma^\sharp[v \mapsto z^\sharp], \Theta)
\end{aligned}$$

Figure 4 Abstract semantics.

Abstract Semantics. Let $\mathbb{Z}^\sharp$ be an abstract domain for $2^\mathbb{Z}$, e.g., intervals, and $\mathbb{M}^\sharp \subseteq^\sharp \mathbb{Z}^\sharp$ its subdomain for $2^\mathbb{M}$. Consider the non-relational abstract semantics partially shown in Figure 4. The evaluation of an expression e is now performed in an abstract environment $\sigma^\sharp$ yielding an abstract value for e and a set of all safety checks performed. Similarly, the execution of a statement s in an abstract environment $\sigma^\sharp$ yields an abstract environment after executing s and a set of all safety checks performed. Again, we only spell out the semantics for some kinds of expressions and statements, and for simplicity, the provided semantics for addition does not refine the summands in the abstract environment to non-overflowing abstract values. Like the concrete semantics, the returned checks are computed separately and will include checks even if the abstract value is $\bot^\sharp$ due to a definite runtime error.

▶ **Example 2.** Let us analyze Example 1 with the interval domain. We abstract $\texttt{rand}(0,1)$ as the interval $[0,1]$ and the constant 1 as $[1,1]$. Thus, for any abstract environment $\sigma^\sharp \in \Sigma^\sharp$,

$$\mathbb{E}^\sharp[\![\texttt{rand}(0,1)]\!](\sigma^\sharp) = ([0,1], \emptyset), \qquad \mathbb{E}^\sharp[\![1]\!](\sigma^\sharp) = ([1,1], \emptyset).$$

Then $z_1^\sharp +^\sharp z_2^\sharp = [1,2]$ and (for the above example) the abstract machine-integer range is $\mathbb{M}^\sharp = [-1,1]$. We have $[1,2] \not\subseteq^\sharp \mathbb{M}^\sharp$ (possible overflow) but also $[1,2] \cap^\sharp \mathbb{M}^\sharp = [1,1] \neq \bot^\sharp$ (there is at least one non-overflowing result). Consequently, according to Figure 4, both (possibly empty) singleton comprehensions are enabled:

$$\begin{aligned}
\Theta' &= \{((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), \text{✗}) \mid [1,2] \not\subseteq^\sharp \mathbb{M}^\sharp\} = \{((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), \text{✗})\}, \\
\Theta'' &= \{((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), \text{✓}) \mid [1,2] \cap^\sharp \mathbb{M}^\sharp \neq \bot^\sharp\} = \{((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), \text{✓})\}
\end{aligned}$$

and the abstract evaluation returns

$$\mathbb{E}^\sharp[\![e]\!](\sigma^\sharp) = \Big([1,1], \{((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), \text{✓}),\ ((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), \text{✗})\}\Big).$$

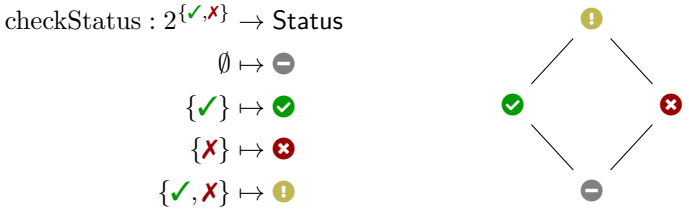


Figure 5 The checkStatus isomorphism and its induced lattice.

Soundness. The abstract semantics of expressions and statements are *sound* w.r.t. the concrete semantics if

$$\forall e \in \mathsf{Expr}, \sigma^\sharp \in \Sigma^\sharp, \sigma \in \gamma_\Sigma(\sigma^\sharp).$$
$$\mathbb{E}^\sharp[\![e]\!](\sigma^\sharp) = (z^\sharp, \Theta^\sharp) \wedge \mathbb{E}[\![e]\!](\sigma) = (M, \Theta) \implies M \subseteq \gamma_\mathbb{Z}(z^\sharp) \wedge \Theta \subseteq \Theta^\sharp$$

and

$$\forall s \in \mathsf{Stmt}, \sigma^\sharp \in \Sigma^\sharp.$$
$$\mathbb{S}^\sharp[\![s]\!](\sigma^\sharp) = (\sigma'^\sharp, \Theta^\sharp) \wedge \mathbb{S}[\![s]\!](\gamma_\Sigma(\sigma^\sharp)) = (S', \Theta) \implies S' \subseteq \gamma_\Sigma(\sigma'^\sharp) \wedge \Theta \subseteq \Theta^\sharp$$

where the conditions $M \subseteq \gamma_\mathbb{Z}(z^\sharp)$ and $S' \subseteq \gamma_\Sigma(\sigma'^\sharp)$ are the standard soundness statements for the values of e and environments after s, respectively.

Because checks have the same type in the concrete and abstract semantics, the condition $\Theta \subseteq \Theta^\sharp$ encodes check soundness (via an identity concretization). It concisely states the following for all $\varphi \in \mathsf{PO}$:

1. If the check may fail ($(\varphi, ✗) \in \Theta$), then the analysis should say so ($(\varphi, ✗) \in \Theta^\sharp$).
2. If the check is reachable ($(\varphi, ✓) \in \Theta \vee (\varphi, ✗) \in \Theta$), then the analysis should check it ($(\varphi, ✓) \in \Theta^\sharp \vee (\varphi, ✗) \in \Theta^\sharp$).

Post-processing. Note that an analysis can yield different check results at a given program location, e.g., in different loop iterations or function calling contexts. We thus introduce a post-processing step, transforming the analysis checks Θ into analysis results $\sigma_a : \mathcal{L} \times \mathcal{C} \to \mathsf{Status}$. To do so, we define a lattice isomorphism in Figure 5. Then, the post-processed results are given by

$$\sigma_a(\varphi) = \text{checkStatus}(\Theta(\varphi)) \quad \text{where} \quad \Theta(\varphi) = \{x \mid (\varphi, x) \in \Theta\}.$$

Intuitively, the statuses mean the following:

- (unreachable, dead code) means no abstract execution reaches φ,
- (safe, proven, always succeeds, success) means the check at φ always succeeds in all abstract executions,
- (never succeeds, error) means it always fails if reached (and thus is typically reported as an error),
- (unknown, warning) means that both success and failure occur either because of over-approximation within an abstract evaluation or because different contexts or paths result in different outcomes.

Note that we only target traditional sound static analyzers that over-approximate the program semantics, as opposed to bug finders. Thus, ⛔ does not mean a feasible counterexample. Both ⛔ and ❗ are verdicts in the *abstract* semantics and may be false positives caused by imprecise over-approximation (e.g., spurious reachability). However, ⛔ is definitive up to reachability: the check fails in every concrete execution that reaches it, so the program is safe only if the offending code is dead. Thus, this is a useful distinction that sound analyzers make in practice; e.g., Goblint, Mopsa and IKOS [10] also return the above four statuses.

▶ **Example 3.** Recall that for Example 2 we obtained that both $((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), ✓)$ and $((\ell_{\mathsf{ex}}, \mathsf{intOverflow}), ✗)$ are in the set of analysis checks $\Theta^\sharp$. After post-processing, the set of outcomes for the proof obligation $(\ell_{\mathsf{ex}}, \mathsf{intOverflow})$ is $\{✓, ✗\}$ and $\sigma_a((\ell_{\mathsf{ex}}, \mathsf{intOverflow})) =$ ❗.

3 Cross-Analyzer Comparison and Joint Progress

The analysis of real-world C code is difficult, and in practice, we cannot take for granted that the analyzers are always sound or even correctly cover all POs. Thus, we wish to compare and contrast analysis results from multiple analyzers. For a given PO, we may compare the results of two analyzers a_1 and a_2 to obtain the cross-analyzer comparison matrix in Table 1. Interpreting this table reveals several qualitatively different comparison outcomes. These outcomes form a sequence of mutually exclusive cases, applied in the order listed below; once a pair of statuses matches an earlier case, it is not considered by the subsequent ones:

Positive agreement (A^+): Both analyzers report the PO safe or dead (✅ or ⛔).

Negative agreement (A^-): Both analyzers report the PO as potentially unsafe (❗ or ⛔).

Coverage disagreement (D): One analyzer does not report check results for a PO (⛔), while the other does (❗, ✅ or ⛔).

Precision asymmetry (P): One analyzer reports an uncertain outcome (❗), while the other reports a definite outcome (✅ or ⛔).

Contradiction (C): One analyzer reports the PO as safe (✅), and the other as unsafe (⛔).

While the pairwise classification captures agreements and conflicts between two analyzers, it does not directly generalize to sets of more than two analyzers, nor does it provide a notion of collective progress. To address both of these aspects, we rely on a notion of *selectivity*, which allows us to summarize and combine analyzer results in a principled way. For each analyzer a, we consider its set of reachable POs $\{\varphi \mid \sigma_a(\varphi) \neq ⛔\}$, that is, the POs that the analyzer has directly checked. Although unreachable POs can be regarded as safe under soundness assumptions, we exclude them when computing progress metrics.

Table 1 Cross-analyzer comparison matrix.

	$\sigma_{a_2}(\varphi)$			
$\sigma_{a_1}(\varphi)$	❗	⛔	✅	⛔
❗	A^-	P	P	D
⛔	P	A^-	C	D
✅	P	C	A^+	D
⛔	D	D	D	A^+

▶ **Definition 4** (Selectivity [37]). *The selectivity (or discharge rate) of an analyzer a is the ratio of the number of POs it has verified to the total number of POs it has considered:*

$$\mathsf{sel}(a) = \frac{|\{\varphi \mid \sigma_a(\varphi) = \checkmark\}|}{|\{\varphi \mid \sigma_a(\varphi) \neq \ominus\}|}.$$

This notion of selectivity serves as a building block for defining joint progress measures over sets of analyzers and for summarizing their combined behavior beyond pairwise comparison. Given a set of analyzers A, we can compute their optimistic $A^{\sqcap}$ and pessimistic $A^{\sqcup}$ joint results, using the lattice of Figure 5:

$$\sigma_{A^{\sqcap}}(\varphi) = \bigsqcap_{a \in A} \sigma_a(\varphi), \qquad \sigma_{A^{\sqcup}}(\varphi) = \bigsqcup_{a \in A} \sigma_a(\varphi).$$

We define the joint selectivity of a set of analyzers A as the selectivity of their optimistic or pessimistic combinations. For sound tools, we prefer the optimistic joint selectivity $\mathsf{sel}(A^{\sqcap})$ as the measure of joint progress.

The optimistic and pessimistic joint results also provide a partial generalization of the pairwise comparison to sets of more than two analyzers, by applying the above comparison matrix to $\sigma_{A^{\sqcap}}$ and $\sigma_{A^{\sqcup}}$. For example, if $\sigma_{A^{\sqcap}}(\varphi) = \sigma_{A^{\sqcup}}(\varphi)$, we have complete agreement between all analyzers. The precision asymmetries can also be inferred; for example, when $\sigma_{A^{\sqcap}}(\varphi) = \checkmark$ and $\sigma_{A^{\sqcup}}(\varphi) = !$, some analyzers succeeded in proving the PO while others did not. However, we cannot distinguish contradictions from coverage disagreements because in both cases, the joint results would be $\sigma_{A^{\sqcap}}(\varphi) = \ominus$ and $\sigma_{A^{\sqcup}}(\varphi) = !$. Therefore, we currently restrict ourselves to pairwise comparisons when analyzing disagreements between analyzers.

Finally, we are interested in tracking the alignment between analyzers; that is, the extent to which they discover and agree on the same set of POs:

▶ **Definition 5** (Alignment). *The alignment of a set of analyzers A is the ratio of the number of POs for which all analyzers agree on the same status to the total number of POs considered by at least one analyzer:*

$$\mathsf{align}(A) = \frac{|\{\varphi \mid \forall a_1, a_2 \in A, \sigma_{a_1}(\varphi) = \sigma_{a_2}(\varphi)\}|}{|\{\varphi \mid \exists a \in A, \sigma_a(\varphi) \neq \ominus\}|} = \frac{|\{\varphi \mid \sigma_{A^{\sqcap}}(\varphi) = \sigma_{A^{\sqcup}}(\varphi)\}|}{|\{\varphi \mid \sigma_{A^{\sqcup}}(\varphi) \neq \ominus\}|}.$$

Differential Testing Facilitated by Comparison. The cross-analyzer comparison naturally enables a differential testing workflow for static analyzers. By relying on POs and their statuses, it becomes straightforward to apply differential testing [27], in which a program can be analyzed by multiple analyzers, and their outputs systematically compared to identify discrepancies.

In this setting, differential testing is performed manually: the comparison results are inspected to identify disagreements between analyzers. The comparison makes such disagreements explicit and actionable, turning them into systematic opportunities for inspection and improvement. These disagreements may indicate bugs, imprecision, or mismatches in modeling assumptions between analyzers, and, thus, provide concrete cases for improving both analyzer implementations and their PO reporting. We evaluate this approach by extracting disagreement examples in Section 5.3.

4 Open Verification Dashboard

To make the formal comparison model of Section 3 actionable in practice, we developed the Open Verification Dashboard (OVD), a system that aggregates and aligns PO check results

produced by multiple static analyzers. Each static analyzer emits a *proof obligation report* consisting of proof obligations and their associated statuses (✅, ❌, ⚠, ⛔). At a high level, the dashboard consumes such reports produced by individual analyzers and combines them into a single integrated view, presented in Section 4.1.

While the dashboard directly operationalizes the formal comparison model introduced in Section 3, its implementation must account for heterogeneity in how analyzers report POs in practice. In the formal model, POs are abstracted as pairs $\mathcal{L} \times \mathcal{C}$, representing a source location and a safety check category. In practice, however, analyzers differ substantially in the granularity and conventions they use to identify source locations and to classify safety checks. Sections 4.2 and 4.3 describe the design decisions we made to reconcile these differences, refining the abstract notions of locations ($\mathcal{L}$) and categories ($\mathcal{C}$) where necessary to enable meaningful cross-tool comparison. Section 4.4 complements this by outlining how new analyzers can be integrated into OVD.

4.1 Interaction model

The dashboard exposes its functionality through two complementary interfaces: a command-line interface (CLI) and a web-based graphical interface (GUI). Both interfaces operate on a common JSON-based interchange format, which minimizes the integration effort for new analyzers and enables downstream processing by external tools.

The two interfaces serve complementary roles. The CLI is the primary entry point for integrating the dashboard into automated verification workflows. It supports the collection of analyzer outputs, execution of comparisons, and management of dashboard projects in a scriptable manner. The GUI builds on the same underlying data and provides interactive access to verification results, enabling exploration of disagreements and coverage gaps. Beyond visualization, it also supports project management and sharing of results among users with access to the dashboard server.

The global statistics view (Figure 6) provides a quantitative overview of analyzer alignment across the target codebase. This exposes the cardinality of the different sets of PO statuses described in Section 3. Rather than reporting absolute correctness, this view highlights regions of consensus and conflict, allowing us to prioritize inspection effort. The global progress indicator summarizes the proportion of POs on which analyzers agree; full alignment indicates consistency between tools, but not program correctness, which must be assessed at the level of individual files and POs.

The detailed view (Figure 1) shows a finer-grained view by juxtaposing source code alongside the proof obligation report. By default, this view emphasizes conflicts between analyzers, but it can be filtered to display only POs where both analyzers report the same status. This makes semantic disagreements explicit at the level of individual program locations and facilitates targeted investigation and debugging.

4.2 Location alignment

We assumed in our formalization that POs are identified by a unique location from a set of locations $\mathcal{L}$. In practice, analyzers report a *text range* as an ordered pair $p_s : p_e$ representing the start and end positions of the relevant expression in the source code, and there can be slight differences in how analyzers report these ranges:

1. IKOS reports a single column, namely the start of the offending operation. This is most similar to our formalization.

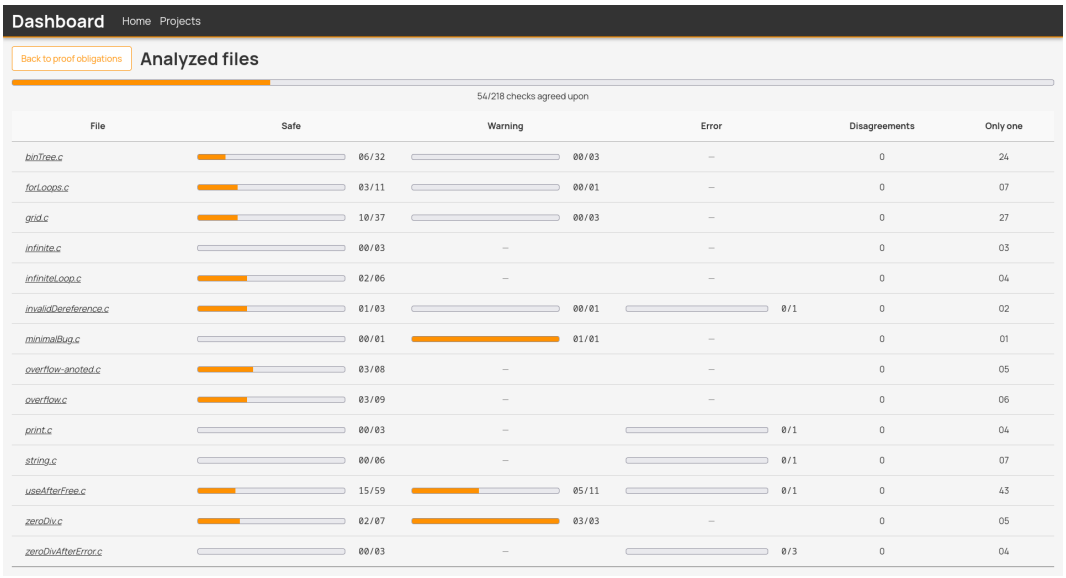


Figure 6 Screenshot of the dashboard GUI global statistics view. Each column presents the number of POs corresponding to the criteria in the header. The first three columns show cases where analyzers agree. The fourth shows the number of POs where there is a conflict, and the last shows the number of POs that are covered by only one analyzer.

2. Goblint reports all warnings at statement granularity in the program because its frontend does not track ranges of individual subexpressions.
3. Mopsa reports the precise range of the offending subexpression, i.e., the subtree in the AST whose root is the offending operation.

We would encourage all tools to report ranges as precisely as possible, but for now we need to handle the differences, and account for the fact that tools are sometimes off-by-one in their reported ranges (e.g., due to preprocessing and frontend transformations affecting location tracking in Goblint). In order to apply our formal comparison model, we need to compute a unified set of proof obligations PO_A for a given set of analyzers $A = \{a_1, a_2, \ldots, a_n\}$. Ideally, this would be the union of all individual proof obligations, $\mathsf{PO}_A = \bigcup_{a \in A} \mathsf{PO}_a$, but in practice, we need to normalize the locations across analyzers. For the union of proof obligations PO_A, we compute an equivalence relation between proof obligations, $\sim \subseteq \mathsf{PO}_A \times \mathsf{PO}_A$. We can then define a result table as the function $\mathcal{R}$ on the canonical representatives of the equivalence classes of proof obligations:

$$\mathcal{R} \colon A \times \mathsf{PO}_A/\sim \longrightarrow \mathsf{Status}, \qquad \mathcal{R}(a, \varphi) = \bigsqcup \{\sigma_a(\varphi_a) \mid \varphi_a \in \mathsf{PO}_a, \varphi_a \sim \varphi\}$$

To define the relation on $\mathsf{PO}_A = \mathcal{L}_A \times \mathcal{C}$, we first normalize locations. We represent each reported location as a source range and define an *overlap* relation

$$\ell_1 \approx_{\mathcal{L}} \ell_2 \iff \ell_1 \cap \ell_2 \neq \emptyset.$$

We then let $\sim_{\mathcal{L}}$ be the transitive closure of $\approx_{\mathcal{L}}$, which groups together all ranges that are connected by overlaps. Based on this relation, we can define the relation $\sim$ on POs as follows:

$$\forall \varphi_1 = (\ell_1, c_1), \varphi_2 = (\ell_2, c_2) : \varphi_1 \sim \varphi_2 \iff \ell_1 \sim_{\mathcal{L}} \ell_2 \wedge c_1 = c_2.$$

Figure 7 Example of different range conventions. In the code, the red rectangle represents a per-statement range and the green dashed rectangles represent per-expression ranges. Then, if analyzer a_1 reports on the entire range, but a_2 reports on the two subexpressions separately, we need to aggregate the results; here, the analyzers are effectively in agreement.

▶ **Example 6.** Consider the example from Figure 7, where we have two analyzers a_1 and a_2 reporting on the same code snippet. Analyzer a_1 reports a single PO on the entire return expression, while analyzer a_2 reports two POs on the two subexpressions separately. Since the ranges overlap, the relation $\sim$ will relate the two POs of analyzer a_2 to the single PO of analyzer a_1, and they will be grouped together in the same equivalence class φ. Then, when we compute the result table $\mathcal{R}$, we will have $\mathcal{R}(a_1, \varphi) = \sigma_{a_1}(\varphi_1) =$ ❗ and $\mathcal{R}(a_2, \varphi) = \sigma_{a_2}(\varphi_2) \sqcup \sigma_{a_2}(\varphi_3) =$ ✅ $\sqcup$ ❌ $=$ ❗.

While transitive closure is generally expensive, in this setting it reduces to computing the union of overlapping intervals (i.e., connected components of the interval intersection graph). We sort the POs primarily by category and secondarily by start location, and compute the equivalence classes efficiently by a one-dimensional sweep-line algorithm [16].

4.3 Category refinement

While our formalization treats $\mathcal{C}$ as a fixed, finite set of abstract safety check categories, the concrete choice of categories is a design decision that impacts the alignment of proof obligation results across analyzers. Initially, we used a single, coarse-grained category for *integer overflow*. However, this category turned out to be too general and ambiguous to precisely capture what is considered integer overflow across C analyzers, their configurations, and usage scenarios. For example, the following distinctions can be made:

- In C, signed integer overflow constitutes undefined behavior (pre-C23), while unsigned overflow wraps around in a well-defined manner (which may still be unintended).
- Implicit casts may occur due to accidental type mismatches, while explicit casts are more intentional.
- Moreover, implicit casts are invisible in the code and arise from a variety of subtle C behaviors (integer promotions, arithmetic conversions, etc.), while explicit casts are clearly visible in code.

In practice, Goblint and Mopsa differed in how they classified overflow checks, which led to a high rate of mismatches when comparing PO results at this coarse level, resulting in poor signal-to-noise for cross-tool comparison. Thus, we adopted a flat but finer-grained classification that distinguishes (i) signed vs unsigned overflow, and (ii) the program construct in which the overflow occurs. This refinement was guided by systematic comparison of analyzer reports and aims to make explicit semantic distinctions already present in the tools that stem from different language constructs in C, while keeping the category system simple and tool-agnostic. Concretely, we split integer overflows into six categories generated by the following regular expression:

(signed/unsigned) integer overflow in (arithmetic operator/explicit cast/implicit cast).

4.4 Integrating a New Analyzer

To make a new analyzer compatible with OVD, the tool provider must expose the PO-level outcomes as used by the transparent static analysis framework. In particular, the tool should report not only alarms or potentially failing checks, but also successful safety checks. Each reported PO is exported in the dashboard interchange format with at least its source range, safety category, and status (✅, ❌, ⚠, or ⛔).

The required integration effort depends on whether the analyzer already records such outcomes. For tools with explicit PO or assertion tracking, integration is largely a matter of JSON export. For tools that only expose final alarms, each analysis operation that may raise an alarm must instead record the corresponding PO. This may require some engineering effort when the code that identifies the POs is not decoupled from the semantic checks of property failures, as was the case for Goblint.

This basic export does not require full cross-tool alignment. Once reports are available, OVD can expose missing checks, mismatched source ranges, and category disagreements, which can then be refined incrementally as in our Goblint/Mopsa integration (see Section 5). The same interface applies to verification tools beyond abstract interpretation, provided that they report PO-level outcomes rather than only a whole-program verdict.

5 Evaluation

In this section, we empirically demonstrate the practical benefits of the proposed dashboard for transparent, fine-grained, PO-level comparison of static analysis results. Transparent static analysis is what makes this comparison more informative than alarm matching: each operation receives one of four possible check statuses (✅, ❌, ⚠, or ⛔). Thus, when analyzer a_1 reports an alarm for an operation and analyzer a_2 does not, OVD can distinguish a check that a_2 proved safe from one that a_2 did not reach or report. We use this information to compare discrepancies and prioritize which ones to investigate.

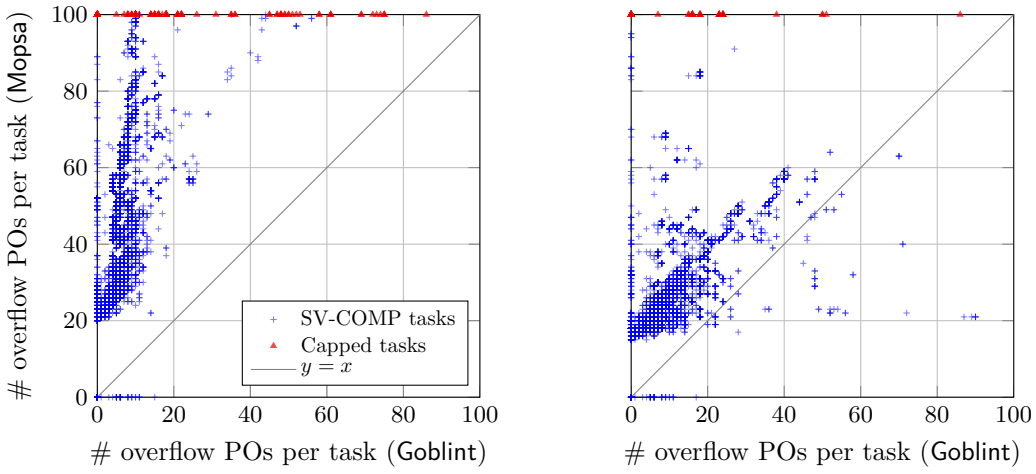
Our evaluation has three goals. First, we validate that POs can be extracted and aligned consistently across analyzers, and show that this alignment can result in tangible precision improvements in analyzer implementation. Second, we quantify cross-tool complementarity and the benefit of combining tools for joint progress. Third, we assess whether PO-guided distillation yields concise and actionable disagreement examples for debugging the differences exposed by the dashboard. These goals motivate the following research questions:

RQ1 (Complementarity). Given a set of proof obligations from multiple programs, how often do Goblint and Mopsa agree, disagree, strictly refine one another (i.e., exhibit precision asymmetry), or directly contradict each other?

RQ2 (Joint Benefit). How much do we gain by considering both tools together, rather than either alone, when scored against SV-COMP oracles, and when used as a coverage oracle on substantially larger coreutils programs?

RQ3 (Distillation). How effectively does the dashboard help us *discover and distill* interesting disagreement examples between Mopsa and Goblint by guiding program reduction with PO-level disagreement conditions, compared to using only global SV-COMP verdicts?

Tools. We evaluate two sound static analyzers for C based on abstract interpretation: Goblint and Mopsa. Both aim to prove the absence of runtime errors by computing sound over-approximations of program behaviors. Each analyzer is instrumented to emit transparent proof obligation reports compatible with the Open Verification Dashboard (OVD).



(a) Before alignment **(b)** After alignment

Figure 8 Per-task comparison of the number of integer overflow proof obligation checks reported by Goblint and Mopsa on SV-COMP. Each point is one task. Darker regions indicate higher densities of tasks due to overlapping points. For readability, tasks with more than 100 Mopsa checks are shown at $y = 100$ (triangles).

Benchmarks and scope. Our evaluation uses the sequential C benchmark tasks from SV-COMP 2026 [8] annotated with the no-overflow property, comprising 8,358 programs. SV-COMP serves as the de facto benchmark suite for program verification, and each task is annotated with a reference verdict: true when the program is known to satisfy a specification, and false when there exists a program execution that violates it.

Accordingly, we restrict our attention to POs corresponding to integer overflow checks in arithmetic operations from the category definitions introduced in Section 4.3. This choice aligns with the SV-COMP overflow track, which focuses on arithmetic overflows and does not consider cast-induced overflows. SV-COMP does include left-shift overflow; however, we do not currently consider it. By concentrating on arithmetic overflows, we obtain a meaningful setting in which to illustrate the benefits of PO-level comparison for complementarity and alignment (RQ1), joint benefit (RQ2), and reduction-based disagreement extraction (RQ3). Note that while our evaluation focuses on integer overflows, our approach is fully generic over C runtime errors detected by static analyzers, including assertion failures, memory safety violations, floating-point issues, and format or argument mismatches.

To obtain PO-level results for integer overflow checks, we rerun both analyzers on the selected benchmarks and extract their PO reports. The official SV-COMP configurations of these tools apply a *portfolio* of successively more expressive analyses to solve each verification task. For our evaluation, we prefer a uniform configuration across all benchmarks, so we use basic configurations with a memory limit of 1 GB and a timeout of 90 s. Out of 8,358 programs, both analyzers successfully produced PO reports for 7,421 tasks; in the remaining cases, at least one analyzer terminated without producing a report (e.g., due to an error, timeout, or memory exhaustion).

PO-level data validation and alignment baseline. Before answering the research questions, we evaluate whether the dashboard can reliably extract and align POs across analyzers on the SV-COMP benchmarks. While SV-COMP benchmarks are widely used for tool

comparison, the suite is designed for the coarse-grained reporting approach of SV-COMP, and consists of many small tasks with potentially few POs. Thus, we need to validate that it is a suitable benchmark suite for our PO-level evaluation. We first measure the number and distribution of POs produced by each analyzer per task, quantifying how much semantic detail the dashboard exposes beyond single task-level verdicts. This fine-grained information is particularly important in real-world verification scenarios, where reference verdicts are unavailable, and verification progress must be assessed based on partial results.

Figure 8 shows, for each SV-COMP task, the number of integer overflow POs produced by Goblint and Mopsa. Figure 8a presents the raw counts prior to alignment, while Figure 8b shows the counts after alignment. The wide spread of points demonstrates that, even for benchmarks with a single task-level verdict, analyzers expose markedly different amounts of PO-level structure. In particular, many tasks give rise to tens of distinct overflow checks, illustrating the amount of semantic detail exposed by the standardized PO reports. Most points lie above the diagonal, as Mopsa reports separate POs for identical program locations reached under different call stacks, whereas Goblint aggregates such cases into a single PO. This baseline also motivates the explicit alignment refinement discussed in RQ1.

5.1 RQ1 (Complementarity)

With RQ1, we explore how often the tools report checks for the same POs and, where they do, how often they assign the same or different statuses to the POs defined by the same location and category. For the analyzer pair (Mopsa, Goblint), we construct the 4×4 comparison matrix of Table 1 from Section 3 for arithmetic-operation POs in the SV-COMP subset. Table 2 shows this cross-classification of PO statuses returned by Mopsa (rows) and Goblint (columns). Diagonal cells correspond to agreements, while off-diagonal cells represent precision asymmetries, coverage disagreements, or direct contradictions as defined in Section 3.

Alignment refinement. A prerequisite for meaningful comparison is reliable alignment of POs across analyzers. We therefore used preliminary comparison results to iteratively refine PO alignment. We first examined POs for which an overflow check was reported by only one analyzer, extracted the corresponding program expressions, and manually investigated the causes of these discrepancies. This process revealed several recurring classes of issues, including spurious checks, missing checks, and mismatches in reported locations (see Table 3). After applying the corresponding fixes, the number of POs reported by only one analyzer decreased by orders of magnitude, substantially improving cross-tool consistency. The pre-alignment matrix (Table 2a) shows the situation prior to alignment fixes and serves as evidence of the high number of unmatched POs.

Comparison results. Together with per-tool selectivity (Section 3), the post-alignment matrix (Table 2b) provides a clear picture of where each analyzer tends to be more decisive, where they corroborate one another, and where they systematically differ. In the post-alignment matrix, the vast majority of POs fall into diagonal cells, indicating strong agreement between the analyzers. The largest cell corresponds to both tools reporting POs as safe (✅), showing substantial overlap in obligations proven safe by both analyzers. Off-diagonal entries are dominated by precision asymmetries (❗ and ✅), suggesting differences in analysis precision rather than semantic conflicts. Direct contradictions (✅ and ❌) are absent, which is consistent with the expected soundness of both tools. Note that the totals differ between

Table 2 Cross-analyzer comparison of arithmetic overflow POs on the SV-COMP subset before and after alignment fixes described in Table 3.

(a) Comparison matrix **before** PO alignment fixes.

$\sigma_{\text{Mopsa}}(\varphi)$ \ $\sigma_{\text{Goblint}}(\varphi)$	!	✗	✓	⊖
!	8,122	0	2,268	159
✗	1	0	0	6
✓	222	0	11,700	23,754
⊖	949	0	1,804	0

(b) Comparison matrix **after** PO alignment fixes.

$\sigma_{\text{Mopsa}}(\varphi)$ \ $\sigma_{\text{Goblint}}(\varphi)$	!	✗	✓	⊖
!	8,123	0	2,255	146
✗	1	0	0	0
✓	222	0	32,226	652
⊖	944	0	2,126	0

Table 3 Issues identified by differential testing of overflow-related POs reported by Mopsa and Goblint.

Issue	Mopsa	Goblint
Spurious check	#251	#1909, #1910, #1932
Missing check		#1933, cil#215, cil#216
Wrong location	#248	cil#211

the two matrices because we do not have a canonical set of POs for each task. Removing spurious checks from a single analyzer will reduce the total number of POs in the comparison.

Precision gains from alignment. Importantly, the alignment process also uncovered cases of unnecessarily conservative behavior within the analyzers. For example, fixing issues in Goblint's internal modeling (e.g., in pointer arithmetic (#1909) and `calloc` handling (#1932)) led to measurable precision gains, increasing Goblint's number of successfully discharged tasks by 13. Notably, these improvements were not the result of targeted precision engineering, but emerged directly from inspecting POs that failed to align with Mopsa. This demonstrates that PO-level comparison is not merely a diagnostic tool for reporting discrepancies, but can actively reveal latent imprecision and lead to concrete improvements in analyzer implementation.

While our evaluation focuses on arithmetic-overflow POs under basic tool configurations, the same alignment methodology applies to other safety categories, properties and configurations. Preliminary experiments on the SV-COMP memory safety track (not reported here) further support the PO-generic nature of our approach. The results presented here already demonstrate that even in this restricted setting, transparent PO-level comparison yields substantial practical benefits.

5.2 RQ2 (Joint Benefit)

Using the same PO reports extracted for RQ1, RQ2 investigates how Mopsa and Goblint *jointly* contribute useful information about POs. In particular, we assess how much they complement each other toward joint verification of a program. We consider the optimistic combination ($A^{\sqcap}$) as defined in Section 3 and ask whether there exist SV-COMP tasks for which the tools verify different subsets of POs such that they jointly verify the entire set. We found no SV-COMP task that is not verified by either Mopsa or Goblint individually but becomes fully verified with their optimistic combination. Thus, under the configurations

Table 4 Examples of joint progress matrices illustrating the optimistic combination. Each cell shows the number of POs classified by the status pair returned by Mopsa (rows) and Goblint (columns).

(a) `bresenham-ll_valuebound1.c`.

$\sigma_{\text{Mopsa}}(\varphi)$ \ $\sigma_{\text{Goblint}}(\varphi)$	!	✗	✓	⊖
!	2	0	**3**	0
✗	0	0	0	0
✓	**1**	0	5	0
⊖	0	0	0	0

(b) `uniq_3args_ok.c`.

$\sigma_{\text{Mopsa}}(\varphi)$ \ $\sigma_{\text{Goblint}}(\varphi)$	!	✗	✓	⊖
!	16	0	**3**	0
✗	0	0	0	0
✓	**8**	0	46	935
⊖	2	0	14	0

used in this evaluation, combining the tools does not yield additional fully verified programs.

At the task level, the tools exhibit a measurable degree of complementarity in SV-COMP scoring. With their official SV-COMP 2026 portfolio configurations [38, 42], Mopsa verified 187 C.NoOverflows tasks that Goblint did not verify, while Goblint verified 174 tasks that Mopsa did not, indicating distinct strengths across the benchmark suite. However, this complementarity does not carry over to the PO-level, as we observed no benchmark where the optimistic combination suffices to verify all remaining obligations. Still, limited PO-level complementarity is observable. There are 11 tasks on which the optimistic combination verifies strictly more POs than either tool alone. Among these, 8 tasks have verdict true, while 3 have verdict false. However, the 8 true tasks correspond to variants of the same benchmark differing only in loop bounds. In none of the 8 cases does the union suffice to verify all POs.

Representative examples are shown in Table 4. Each joint progress matrix cross-classifies the statuses assigned by Mopsa (rows) and Goblint (columns), highlighting POs where one analyzer proves safety (✓) while the other fails to do so (!). These off-diagonal cells correspond precisely to the additional obligations discharged by the optimistic combination.

To quantify the *joint benefit* of using both analyzers together, we consider the selectivity of the combined analysis ($\mathsf{sel}(A^{\sqcap})$), and the individual selectivities of each analyzer. For instance, for the benchmark in Table 4a, the selectivity of Mopsa is 6/11, the selectivity of Goblint is 8/11, and the selectivity of the combined analysis is 9/11.

We observed similar patterns on substantially larger coreutils programs when running both analyzers with increased resource limits (450 s and 2 GB memory). The programs (`cut`, `env`, and `uniq`) each comprise over 200,000 lines of C code. On these programs, the optimistic combination consistently improves selectivity compared to either tool alone. For instance, on `uniq`, Mopsa verifies 54 out of 73 overflow POs reported by both tools, Goblint verifies 49, while their optimistic combination verifies 57 (see Table 4b). Similar improvements are observed on the other two programs. Although the optimistic combination does not yet discharge all remaining obligations, the remaining POs indicate where additional precision improvements could enable complete joint verification.

5.3 RQ3 (Distillation): Extracting Disagreement Examples

RQ1 and RQ2 use dashboard comparisons to quantify where analyzers agree, diverge, or complement each other. Complex C programs can make it difficult to understand the root

cause of the remaining disagreements between analyzers. To address this, we combine differential testing with automated test-case minimization [41]. We write $\mathsf{cvise}(P_0, \lambda P.\, x)$ for invoking cvise on the initial program P_0 with an interestingness predicate over candidate programs P; a candidate is kept when the expression x evaluates to true. A minimizer starts from an input program and repeatedly tries to simplify it, keeping a candidate only when an interestingness test still succeeds. In our setting, the test compiles the candidate, reruns both analyzers, and checks that the reduced program still exposes the disagreement we want to study. This oracle-guided reduction is performed with cvise [31], a recent variant of C-Reduce [12]. The goal is to extract small, human-readable disagreement examples that are also well-suited as regression tests.

Reduction objectives. Let $A = \{a_1, a_2\}$ be the two analyzers under comparison and let P_0 be the original program. We first run both analyzers on P_0, obtaining baseline results $r_i^0 = \mathsf{run}(a_i, P_0)$, and instantiate one of two interestingness predicates for cvise.

Verdict objective. The verdict strategy preserves only the baseline task-level verdict pair $(v_1^\star, v_2^\star)$, where $v_i^\star = \mathsf{verdict}(r_i^0)$. For our sound tools, these baseline verdicts are true or unknown. Thus, we call $\mathsf{cvise}(P_0, \lambda P.\, \mathsf{isInteresting}_{\mathsf{ver}}(P; v_1^\star, v_2^\star))$, where

$$\begin{aligned}\mathsf{isInteresting}_{\mathsf{ver}}(P; v_1^\star, v_2^\star) = \ & \textbf{if } \neg\mathsf{compiles}(P) \textbf{ then } \mathsf{false} \textbf{ else} \\ & \textbf{let } r_1 = \mathsf{run}(a_1, P) \textbf{ and } r_2 = \mathsf{run}(a_2, P) \textbf{ in} \\ & \mathsf{verdict}(r_1) = v_1^\star \wedge \mathsf{verdict}(r_2) = v_2^\star.\end{aligned}$$

Dashboard objective. The dashboard strategy preserves a directional PO discrepancy. Recall $\mathsf{Status} = \{\ominus, \checkmark, \times, !\}$, and let $\mathcal{R}_0 \colon A \times \mathsf{PO}_A/\!\sim \to \mathsf{Status}$ be the aligned dashboard table for the baseline run. We select an obligation $\varphi^\star \in \mathsf{PO}_A/\!\sim$, write its category as $c^\star$, and keep the directional status pair $(s_1^\star, s_2^\star)$ recorded by $\mathcal{R}_0$, where $s_i^\star \in \mathsf{Status}$. The reduced program may realize this discrepancy on a different obligation of the same category. Thus, we call $\mathsf{cvise}(P_0, \lambda P.\, \mathsf{isInteresting}_{\mathsf{dash}}(P; c^\star, s_1^\star, s_2^\star))$, where

$$\begin{aligned}\mathsf{isInteresting}_{\mathsf{dash}}(P; c^\star, s_1^\star, s_2^\star) = \ & \textbf{if } \neg\mathsf{compiles}(P) \textbf{ then } \mathsf{false} \textbf{ else} \\ & \textbf{let } r_1 = \mathsf{run}(a_1, P) \textbf{ and } r_2 = \mathsf{run}(a_2, P) \textbf{ in} \\ & \textbf{let } \mathcal{R} = \mathsf{dashboard}(r_1, r_2) \textbf{ in} \\ & \exists \varphi \in \mathsf{PO}_A/\!\sim \textbf{ such that } \varphi = (\ell, c^\star) \wedge \\ & \quad \mathcal{R}(a_1, \varphi) = s_1^\star \wedge \mathcal{R}(a_2, \varphi) = s_2^\star.\end{aligned}$$

Qualitative evaluation of distillation during development. During dashboard development, we applied the dashboard objective to programs exhibiting analyzer discrepancies in order to isolate minimal examples triggering a disagreement. An example of such a minimized program is shown in Figure 1.

While code reduction does not guarantee preservation of the exact original conflict, restricting the oracle to the same category has proven sufficient in practice. In all cases we encountered, fixing the issue exposed by the reduced example also resolved the original discrepancy in the larger input program. The issues distilled from the reductions are summarized in Table 5, from which we highlight some concrete examples below.

For instance, the minimal example of only 4 lines in Mopsa issue #234 was extracted from the SV-COMP benchmark `uthash_BER_test5-3.i`, by reducing an original input of 765 lines. Similarly, the example corresponding to Goblint issue #1767 was reduced from 32 lines to 16 lines. Although the original program in this case was already relatively small, the reduction process still proved useful by eliminating syntactically and semantically irrelevant

Table 5 Overview of the bugs found by differential testing.

Issue ID	Analyzer	Description
#234	Mopsa	Spurious invalid memory access report
#1767	Goblint	Unsigned underflow from thread return value cast
#1798	Goblint	Missing increment-overflow warning with interval sets
#1801	Goblint	Integer overflow triggered by memOutOfBounds analysis
#1802	Goblint	Constant string literals treated as top pointers

Table 6 Folder-level averages for RQ3 reduction comparison. The dashboard strategy uses preservation as its criterion; thus, its preservation rate was 100% as expected.

	Task Metrics			Verdict-Based			Dashboard-Based	
Folder	#	Lines	Conflicts	Final lines	Pres. (%)	Time (s)	Final lines	Time (s)
`float-benchs`	6	196.17	1.00	8.67	100.00	184.50	8.67	233.17
`nla-digbench`	4	42.25	5.75	5.00	0.00	104.25	6.00	123.00
`termination-crafted`	11	26.18	1.55	3.73	18.18	119.73	5.91	156.45
`termination-crafted-lit`	11	27.18	1.73	5.27	45.45	92.00	7.00	102.73
`termination-numeric`	3	30.00	1.00	4.00	0.00	124.00	5.67	190.00
`termination-restricted-15`	13	21.62	1.00	4.31	7.69	83.62	8.77	98.62

details. Across these examples, reduction isolates the core construct responsible for the analyzer disagreement, removing surrounding code that does not contribute to the issue. This illustrates the benefit of PO-guided reduction for deriving concise examples.

Systematic comparison methodology. We then systematically investigate whether the simpler verdict-based objective would have been as effective for disagreement extraction as the dashboard objective. We consider SV-COMP tasks annotated with the no-overflow property for which the official SV-COMP 2026 results report clear verdict disagreements (true/unknown or unknown/true). Since we are not using the SV-COMP portfolio configurations, we rerun each task locally and keep only those whose local baseline preserves the expected verdict pair and corresponding dashboard conflict direction. We also require that the baseline runs complete in less than a minute since the reduction needs to run the analyzers multiple times on each task. We were left with 56 tasks containing a disagreement where Goblint reports the overflow check safe, but Mopsa reports it as a warning. This gives both objectives the same starting point: a task-level verdict disagreement and a specific PO-level disagreement direction.

For each minimized file, we record original/final line counts, original/final directional dashboard disagreement counts, and wall-clock reduction time. For clarity, we aggregate results by folder, and for this overview, include only folders containing at least three tasks to keep the table concise. For each folder, we report the number of tasks, average original lines, average original disagreement count, verdict-strategy average final lines, verdict-strategy disagreement preservation rate, and average runtime for both strategies.

Results. Table 6 shows a clear trade-off in this controlled follow-up. The verdict strategy usually produces smaller and faster reductions, but often drops the directional conflict: its preservation rate ranges from 0% to 45.45% on five folders, and reaches 100% only on `float-benchs`. Averaged over all 56 tasks, the verdict strategy yields 5.36 final lines and 112.86 s on average, versus 7.54 lines and 142.30 s for the dashboard strategy. The dashboard

strategy is therefore less aggressive in reducing size, but substantially more reliable for preserving disagreement semantics.

5.4 Threats to Validity

The central claim of our work, and what we aim to substantiate with the evaluation, is that transparent tracking of PO statuses has practical benefits for cross-tool comparison and improvement. Several threats can affect the strength of this inference from our empirical results.

The *internal validity* of our research design depends on the reliability of the alignment process. This process relies on additional code for exporting proof obligations, as well as the dashboard's post-processing and matching logic, which is non-trivial and can itself introduce errors. We mitigated this threat through iterative validation: mismatches were inspected, minimized, and traced to concrete root causes, leading to fixes in both analyzers and in the alignment procedure (Table 3). The fixes to our analyzers were validated against the SV-COMP benchmarks and their reference verdicts.

One may also question the *construct validity* of our comparison metrics (e.g., selectivity and alignment), that is, the extent to which they capture verification progress and cross-tool agreement. The main concern is that these metrics are defined over the extracted set of POs and therefore depend on the granularity and categorization of those obligations. We mitigate this threat by (i) mapping both tools to a shared overflow taxonomy (Section 4.3), and (ii) aggregating overlapping ranges when tools report at different syntactic granularities (Section 4.2). This overlap aggregation is conservative and reports a conflict whenever tools assign different statuses to any overlapping POs. Nevertheless, users should interpret 100% selectivity relative to the POs jointly identified by the tools: if both tools miss the same PO, the program is not verified despite complete selectivity and alignment.

Regarding *external validity*, our evaluation is limited to two analyzers (Goblint and Mopsa). We chose them because they are sound C abstract interpreters active in SV-COMP and already expose whole-program PO-level outcomes. Even for such similar tools, transparent comparison requires alignment and debugging. The first OVD reports exposed missing checks, category mismatches, source-location differences, and implementation issues; investigating them led to 14 fixes in Goblint and Mopsa. Shallow prototype integrations of more tools would not provide comparable evidence. Preliminary testing with IKOS [10] showed that it exposes PO-level information that can be mapped to OVD, but it has not gone through the same alignment process. Frama-C's Eva plug-in would be another natural candidate, but would require a separate native integration effort from the Frama-C team. Plausible candidates for further integration also include SV-COMP participants such as CBMC [14, 29] and UAutomizer [22, 4], whose current outputs partially expose PO information. For example, CBMC reports some checks in execution logs, and UAutomizer identifies checks internally. However, this information is not systematically exposed in a form directly usable by OVD: CBMC provides only coarse source locations (e.g., line numbers), and UAutomizer does not report corresponding program locations for its checks. We also focus on integer overflows, and within SV-COMP primarily on the no-overflow property. This scope provides a large benchmark set with reference verdicts and lets us refine PO categories in a controlled setting. Note, however, that our approach is not overflow-specific: the framework supports arbitrary safety categories, and we expect similar benefits for other runtime errors (e.g., invalid memory accesses, division by zero), but the corresponding category taxonomy and alignment rules would need to be extended and validated. Finally, some SV-COMP tasks can be small and synthetic. To partially mitigate this, we also evaluated the optimistic combination as a

coverage oracle on three larger coreutils programs (RQ2).

For *conclusion validity*, the precise number of verified POs is sensitive to tool configurations and resource limits. While the statistical power of the experiments is limited, our empirical results provide evidence for the expected benefits of transparent reporting. We consider the strongest evidence for the usefulness of our approach to be the qualitative improvements in analyzer implementation that emerged directly from the alignment process and the PO-guided distillation of concrete disagreement examples.

6 Related Work

Reporting and exchanging static analysis results. Textbook definitions of static analyses [35, 33] focus on computing reachable states, leaving implicit the individual semantic checks that were proved or left inconclusive. Transparent reporting in Mopsa exposes these checks to support automated precision metrics [37]; TSA formalizes this behavior and makes POs the basis for soundness and cross-tool comparison.

The Static Analysis Results Interchange Format (SARIF [40]) is an industry-standard JSON schema for reporting issues from static analysis tools. Many modern analyzers and security scanners, including Coverity [5], ESLint [18], and CodeQL [2], can emit SARIF, and viewers such as the VS Code SARIF plugin [32] and GitHub code scanning UI [20] aggregate these results. SARIF is useful for transporting alarms, but it does not provide semantic classifications for alarms or successful checks.

Definitions of witnesses have been developed over the years within the SV-COMP community [6, 3] as a way to validate analyses performed by complex tools. Recent work [44] introduced techniques to guide static analyzers using these witnesses, thereby obtaining better precision than without them. Witnesses contain information about how the analyzer reached a certain conclusion, such as a counterexample trace for a bug report or an inductive invariant for a proof. Comparing and visualizing witnesses is an interesting research direction in its own right.

Comparing static analyzers. In SV-COMP [7], static analyzers have to provide coarse-grained results for a program verification task (deciding whether a program property holds or not). SV-COMP benchmarks require a known ground truth per benchmark. Our broader target is open-ended settings (e.g., a new codebase with no prior oracle), although we also use SV-COMP when reference verdicts are useful for evaluation. In both settings, our comparison is finer-grained: each program analysis reports every semantic check performed. The same granularity also supports selectivity-style precision summaries [37]. Even finer-grained approaches have also focused on measuring the size of abstract domains [47] or, more generally, on quantifying imprecision in static analyses [11]. To the best of our knowledge, these approaches have not been used in experimental settings to compare static analysis frameworks.

Both Mopsa [37] and clam [21] provide internal diffing tools that can compare analysis results across different configurations, determining if one analysis is strictly more precise than another or if any alarms or assumptions differ. These tools do not aim to unify multiple analyzers' outputs for a user. CodeChecker [24] is a tool that can run multiple industrial analyzers (Clang Static Analyzer, Infer) and then presents all their warnings on a unified web interface. Its focus is on aggregating defect reports, not comparing results or building collaborative proofs of correctness.

Differential testing and static analyzers. Our work also combines well with other comparative techniques. Test-case reduction tools like C-Reduce [41] automate the minimization of programs while preserving a property of interest, which in our case is the disagreement between tools. Andreasen et al. [1] show that other test-case reduction techniques, such as delta-debugging, can be used to increase static analysis precision. Our proposed dashboard can aid such debugging scenarios by quickly highlighting the exact program points where tools diverge or struggle. One key requirement in work on soundness and precision testing of static analyzers [27] is to synthesize checks to compare analyzers' results. We believe the TSA framework naturally incorporates important semantics-based checks and would thus simplify differential testing approaches.

User interfaces for verification tools. This work was inspired by other tools with advanced user interfaces for static analysis. Frama-C EVA [15] (Evolved Value Analysis) is the default value analysis plug-in of the Frama-C framework. It provides a rich GUI where the C source code is annotated with alarms; potential runtime errors that value analysis cannot prove safe are shown to the user. The user can interactively inspect the proof obligations and apply other Frama-C plug-ins or manual proofs to discharge them. This interface allows the tracking of proof obligations that remain after the value analysis, but does not produce an exhaustive list of what the value analysis itself could handle. Frama-C's dashboard is also limited to its own plug-ins, whereas we aim to integrate multiple independent analyzers.

Several recent works have also addressed debugging static analyzers [17], combining concrete and abstract debuggers [34], and providing abstract debugger interfaces [23, 37, 26], allowing users to better understand why certain warnings were or were not produced.

Deductive verification. Beyond automated static analysis, there have been efforts to unify results from multiple verification tools. The Why3 platform [19] provides an IDE for deductive program verification, where each proof obligation can be tried by multiple automated provers. The Why3 IDE displays a table of provers and POs, indicating which prover succeeded or failed on each proof obligation. This kind of dashboard combining multiple theorem provers is precisely the inspiration for our proposed verification dashboard. The challenge for us is that automated static analyzers handle verification condition generation and their discharge in a monolithic way, so we reconstruct and correlate the POs from their internal checks.

7 Conclusion

Comparing static analyzers is a difficult task due to the alarm-based reporting traditionally used by analysis tools. In this work, we have formalized a novel transparent static analysis framework reporting not only alarms but every semantic check performed by an analyzer. This framework enables a fine-grained comparison of static analysis results, further helped through the implementation of an Open Verification Dashboard, which takes care of aligning checks between tools with different outputs. We have experimentally evaluated the use of the Open Verification Dashboard on two C static analyzers participating in SV-COMP and identified 14 bugs whose fixes improved both precision and reporting quality.

The PO alignment work paves the way towards a standard for reporting C runtime errors detected by static analyzers. We created and implemented a classification for alarms related to integer overflows. Note that our approach is not restricted to integer overflows. We have also been able to compare analyzers on the memory safety track of SV-COMP, opening the door to additional alignment and classification work in the future. Future dashboard work

could also use sound non-statistical alarm clustering [30] to reduce the number of comparisons shown to users. On the other hand, the dashboard should expose more detail when analyzers intentionally distinguish checks using call-stack or other path-sensitive information, as in Goblint's digest-based analyses [45, 46].

We hope this article makes a case for other static analyzers to implement transparent reporting, which would pave the way for comparisons between other static analysis tools. In preliminary experiments, we have been able to confirm that IKOS [10] already fits our transparent reporting framework and could thus benefit from alignment improvements pinpointed by OVD. We envision that this dashboard can be used in collaborative static analysis efforts to assess which parts of a given software project have been analyzed in depth, and where coverage gaps highlight the need for further investigations.

References

1 Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In Karim Ali and Cristina Cifuentes, editors, *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, pages 31–36. ACM, 2017. doi:10.1145/3088515.3088521.

2 Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, Rome, Italy, July 18-22, 2016*, LIPIcs, pages 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2016.2, doi:10.4230/LIPICS.ECOOP.2016.2.

3 Paulína Ayaziová, Dirk Beyer, Marian Lingsch Rosenfeld, Martin Spiessl, and Jan Strejcek. Software verification witnesses 2.0. In Thomas Neele and Anton Wijs, editors, *Model Checking Software - 30th International Symposium, SPIN 2024, Luxembourg City, Luxembourg, April 8-9, 2024, Proceedings*, volume 14624 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2024. doi:10.1007/978-3-031-66149-5_11.

4 Manuel Bentele, Max Barth, Marcel Ebbinghaus, Jan Körner, Daniel Dietsch, Matthias Heizmann, Dominik Klumpp, Frank Schüssele, and Andreas Podelski. Ultimate automizer with a one-dimensional memory model - (competition contribution). In Sebastian Junges and Guy Katz, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 32nd International Conference, TACAS 2026, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2026, Turin, Italy, April 11-16, 2026, Proceedings, Part II*, Lecture Notes in Computer Science, pages 589–594. Springer, 2026. doi:10.1007/978-3-032-22749-2_37.

5 Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010. doi:10.1145/1646353.1646374.

6 Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. Correctness witnesses: exchanging verification results between verifiers. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 326–337. ACM, 2016. doi:10.1145/2950290.2950351.

7 Dirk Beyer and Jan Strejcek. Improvements in software verification and witness validation: SV-COMP 2025. In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2025), Part III*, volume 15698 of *Lecture Notes in Computer Science*, pages 151–186. Springer, 2025. doi:10.1007/978-3-031-90660-2_9.

8 Dirk Beyer and Jan Strejcek. Evaluating software verifiers for C, Java, and SV-LIB - (report on SV-COMP 2026). In Sebastian Junges and Guy Katz, editors, *Tools and Algorithms for the*

Construction and Analysis of Systems - 32nd International Conference, TACAS 2026, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2026, Turin, Italy, April 11-16, 2026, Proceedings, Part II, Lecture Notes in Computer Science, pages 461–502. Springer, 2026. doi:10.1007/978-3-032-22749-2_23.

9 Sandrine Blazy, David Bühler, and Boris Yakobowski. Structuring abstract interpreters through state and value abstractions. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 112–130. Springer, 2017. doi:10.1007/978-3-319-52234-0_7.

10 Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. IKOS: A framework for static analysis based on abstract interpretation. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, Lecture Notes in Computer Science, pages 271–277. Springer, 2014. doi:10.1007/978-3-319-10431-7_20.

11 Marco Campion, Caterina Urban, Mila Dalla Preda, and Roberto Giacobazzi. A formal framework to measure the incompleteness of abstract interpretations. In Manuel V. Hermenegildo and José F. Morales, editors, *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings*, volume 14284 of *Lecture Notes in Computer Science*, pages 114–138. Springer, 2023. doi:10.1007/978-3-031-44245-2_7.

12 Yang Chen, Eric Eide, and John Regehr. csmith-project/creduce, July 2025. original-date: 2012-02-28T15:55:07Z. URL: https://github.com/csmith-project/creduce.

13 Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 332–343. ACM, 2016. doi:10.1145/2970276.2970347.

14 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, Lecture Notes in Computer Science, pages 168–176. Springer, 2004. doi:10.1007/978-3-540-24730-2_15.

15 Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, Lecture Notes in Computer Science, pages 233–247. Springer, 2012. doi:10.1007/978-3-642-33826-7_16.

16 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Line Segment Intersection*, pages 19–43. Springer, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-77974-2_2.

17 Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. Debugging static analysis. *IEEE Trans. Software Eng.*, 46(7):697–709, 2020. doi:10.1109/TSE.2018.2868349.

18 ESLint Team and Nicholas C. Zakas. ESLint: The pluggable JavaScript linter. https://eslint.org/, 2025. Version 9.29.0, accessed 1 Jul 2025.

19 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, Lecture Notes in Computer Science, pages 125–128. Springer, 2013. doi:10.1007/978-3-642-37036-6_8.

20 GitHub. Uploading static analysis results as SARIF to GitHub code scanning. https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/uploading-a-sarif-file-to-github, 2025. GitHub Docs, accessed 1 Jul 2025.

21 Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, Lecture Notes in Computer Science, pages 343–361. Springer, 2015. doi:10.1007/978-3-319-21690-4_20.

22 Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, Lecture Notes in Computer Science, pages 36–52. Springer, 2013. doi:10.1007/978-3-642-39799-8_2.

23 Karoliine Holter, Juhan Oskar Hennoste, Patrick Lam, Simmo Saan, and Vesal Vojdani. Abstract debuggers: Exploring program behaviors using static analysis results. In Jonathan Edwards and Marcel Taeumel, editors, *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2024, Pasadena, CA, USA, October 23-25, 2024*, pages 130–146. ACM, 2024. doi:10.1145/3689492.3690053.

24 Gábor Horváth, Réka Kovács, Richárd Szalay, Zoltán Porkoláb, Gergely Orbán, and Dániel Krupp. Static code analysis with CodeChecker. *CoRR*, abs/2408.02220, 2024. URL: https://doi.org/10.48550/arXiv.2408.02220, arXiv:2408.02220, doi:10.48550/ARXIV.2408.02220.

25 Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, Lecture Notes in Computer Science, pages 1–18. Springer, 2019. doi:10.1007/978-3-030-41600-3_1.

26 Nat Karmios, Sacha-Élie Ayoun, and Philippa Gardner. Gillian debugging: Swinging through the (compositional symbolic execution) trees. In Sebastian Junges and Guy Katz, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 32nd International Conference, TACAS 2026, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2026, Turin, Italy, April 11-16, 2026, Proceedings, Part II*, Lecture Notes in Computer Science, pages 195–214. Springer, 2026. doi:10.1007/978-3-032-22749-2_10.

27 Christian Klinger, Maria Christakis, and Valentin Wüstholz. Differentially testing soundness and precision of program analyzers. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 239–250. ACM, 2019. doi:10.1145/3293882.3330553.

28 Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, Lecture Notes in Computer Science, pages 461–478, 2016. doi:10.1007/978-3-319-47166-2_32.

29 Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, Lecture Notes in Computer Science, pages 389–391. Springer, 2014. doi:10.1007/978-3-642-54862-8_26.

30 Woosuk Lee, Wonchan Lee, Dongok Kang, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Sound non-statistical clustering of static analysis alarms. *ACM Trans. Program. Lang. Syst.*, 39(4):16:1–16:35, 2017. doi:10.1145/3095021.

31 Martin Liška. marxin/cvise, August 2025. original-date: 2020-04-18T13:04:05Z. URL: https://github.com/marxin/cvise.

32 Microsoft SARIF Viewer Extension Team. SARIF viewer extension for Visual Studio Code. https://marketplace.visualstudio.com/items?itemName=MS-SarifVSCode.sarif-viewer, 2025. VS Code Marketplace listing, accessed 1 Jul 2025.

33 Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Found. Trends Program. Lang.*, 4(3-4):120–372, 2017. doi:10.1561/2500000034.

34 Mats Van Molle, Bram Vandenbogaerde, and Coen De Roover. Cross-level debugging for static analysers. In João Saraiva, Thomas Degueule, and Elizabeth Scott, editors, *SLE*, pages 138–148. ACM, 2023. doi:10.1145/3623476.3623512.

35 Anders Møller and Michael I. Schwartzbach. Static program analysis, June 2024. Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/.

36 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Mopsa-C: Modular domains and relational abstract interpretation for C programs (competition contribution). In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2023), Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 565–570. Springer, 2023. doi:10.1007/978-3-031-30820-8_37.

37 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Easing maintenance of academic static analyzers. *Int. J. Softw. Tools Technol. Transf.*, 26(6):673–686, 2024. URL: https://doi.org/10.1007/s10009-024-00770-1, doi:10.1007/S10009-024-00770-1.

38 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Mopsa-C with trace partitioning and autosuggestions (competition contribution). In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part III*, Lecture Notes in Computer Science, pages 229–235. Springer, 2025. doi:10.1007/978-3-031-90660-2_17.

39 Tukaram Muske and Alexander Serebrenik. Survey of approaches for postprocessing of static analysis alarms. *ACM Comput. Surv.*, 55(3):48:1–48:39, 2023. doi:10.1145/3494521.

40 OASIS. *Static Analysis Results Interchange Format (SARIF) — Version 2.1.0*, 2020. URL: https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html.

41 John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346. ACM, 2012. doi:10.1145/2254064.2254104.

42 Simmo Saan, Ali Rasim Kocal, Michael Petter, Karoliine Holter, Julian Erhard, Michael Schwarz, Vesal Vojdani, and Helmut Seidl. Goblint: A portfolio for mixed flow-sensitive abstract interpretation - (competition contribution). In Sebastian Junges and Guy Katz, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 32nd International Conference, TACAS 2026, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2026, Turin, Italy, April 11-16, 2026, Proceedings, Part II*, Lecture Notes in Computer Science, pages 516–522. Springer, 2026. doi:10.1007/978-3-032-22749-2_26.

43 Simmo Saan, Michael Schwarz, Kalmer Apinis, Julian Erhard, Helmut Seidl, Ralf Vogler, and Vesal Vojdani. Goblint: Thread-modular abstract interpretation using side-effecting constraints - (competition contribution). In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, Lecture Notes in Computer Science, pages 438–442. Springer, 2021. doi:10.1007/978-3-030-72013-1_28.

44 Simmo Saan, Michael Schwarz, Julian Erhard, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. Correctness witness validation by abstract interpretation. In Rayna Dimitrova, Ori Lahav,

and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part I*, volume 14499 of *Lecture Notes in Computer Science*, pages 74–97. Springer, 2024. doi:10.1007/978-3-031-50524-9_4.

45 Michael Schwarz and Julian Erhard. The digest framework: concurrency-sensitivity for abstract interpretation. *Int. J. Softw. Tools Technol. Transf.*, 26(6):727–746, 2024. URL: https://doi.org/10.1007/s10009-024-00773-y, doi:10.1007/S10009-024-00773-Y.

46 Michael Schwarz and Julian Erhard. Data race detection by digest-driven abstract interpretation. In Yu-Fang Chen, Thomas P. Jensen, and Ondrej Lengál, editors, *Verification, Model Checking, and Abstract Interpretation - 27th International Conference, VMCAI 2026, Rennes, France, January 12-13, 2026, Proceedings*, Lecture Notes in Computer Science, pages 309–334. Springer, 2026. doi:10.1007/978-3-032-15700-3_15.

47 Pascal Sotin. Quantifying the Precision of Numerical Abstract Domains. Research report, INRIA Grenoble Rhônes-Alpes, February 2010. URL: https://inria.hal.science/inria-00457324.

48 Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the Goblint approach. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 391–402. ACM, 2016. doi:10.1145/2970276.2970337.