Reusing Caches and Invariants for Efficient and Sound Incremental Static Analysis

Mamy Razafintsialonina
Université Paris-Saclay, CEA, List, Palaiseau, France
Sorbonne Université, CNRS, LIP6, Paris, France

David Bühler
Université Paris-Saclay, CEA, List, Palaiseau, France

Antoine Miné
Sorbonne Université, CNRS, LIP6, Paris, France

Valentin Perrelle
Université Paris-Saclay, CEA, List, Palaiseau, France

Julien Signoles
Université Paris-Saclay, CEA, List, Palaiseau, France

Abstract

Static analysis by means of abstract interpretation is a tool of choice for proving absence of some classes of errors, typically undefined behaviors in C code, in a sound way. However, static analysis tools are hardly integrated in CI/CD processes. One of the main reasons is that they are still time- and memory-expensive to apply after every single patch when developing a program. For solving this issue, incremental static analysis helps developers quickly obtain analysis results after making changes to a program. However, existing approaches are often not guaranteed to be sound, limited to specific analyses, or tied to specific tools. This limits their generalizability and applicability in practice, especially for large and critical software. In this paper, we propose a generic, sound approach to incremental static analysis that is applicable to any abstract interpreter. Our approach leverages the similarity between two versions of a program to soundly reuse previously computed analysis results. We introduce novel methods for summarizing functions and reusing loop invariants. They significantly reduce the cost of reanalysis, while maintaining soundness and a high level of precision. We have formalized our approach, proved it sound, implemented it in Eva, the abstract interpreter of Frama-C, and evaluated it on a set of real-world commits of open-source programs.

2012 ACM Subject Classification Theory of computation → Program reasoning; Software and its engineering → Software verification and validation; Software and its engineering → Formal methods; Software and its engineering → Software notations and tools

Keywords and phrases Abstract Interpretation, Static Analysis, Incremental Analysis

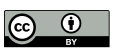
Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.28

Related Version *Extended Version*: `https://hal.science/hal-05110966`

Supplementary Material *Software (Source Code)*: `https://git.frama-c.com/pub/frama-c/-/tree/publi/2025-ecoop-rbmps`
archived at `swh:1:dir:7b4b0787dd0a308bbd0abd211d2f4174224aebba`
Software (ECOOP 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.2.15`

Funding This work was funded by project ANR-22-PECY-0005 "Secureval" managed by the French National Research Agency for France 2030.

Acknowledgements We thank the anonymous reviewers for their valuable feedback, which improved the final version of the paper.



1 Introduction

Abstract interpretation [10, 9] is a sound flow-sensitive static analysis technique that allows the user to prove absence of programming errors, such as undefined behaviors in C code. To be precise enough, it is also usually implemented as a context-sensitive analysis, i.e., the analyzer relies on the function context (such as values of arguments and global variables) when analyzing a function call. However, soundness and preciseness come at a cost: Abstract interpretation can be time- and memory-expensive when analyzing real-world programs. It is not necessarily an issue when analyzing a program once, but it can prevent developers from introducing such a static analysis technique in a CI/CD process to check that their changes do not introduce bugs: Even a seemingly minor syntactic change can have a huge impact on the behavior of the software.

Incremental analyses are one way to reduce the cost of analyzing software after a change by reusing as many results as possible from the analysis of a previous version when analyzing the new version of the code. However, providing a sound flow- and context-sensitive incremental analysis is hard: existing approaches are either unsound [22, 27] or limited to specific classes of analyses [35, 16] or tied to specific analysis tools [1, 33, 28]. This paper presents a generic abstract-interpretation based approach for incremental flow- and context-sensitive analysis which is sound, and supports unbounded memory locations. Our approach is based on soundly reusing analysis results from previous analysis runs and is designed to be is applicable to most existing abstract interpreters. For each type of analysis result to be reused, we provide the conditions under which soundness is guaranteed. Such conditions typically involve program matching, which is the process of determining whether two parts of a program are equivalent or not. We consider two different types of analysis results: Function summaries and loop invariants. For function summaries, we propose a new way to summarize the effect of the analysis of a function in order to avoid re-analyzing the body of a function when the context of the analysis is the same. Our function summaries are soundly reusable and always precise for functions that have not been impacted by the modification of the program. For loop invariants, we propose a way to accelerate the analysis of loops by reusing the loop invariants from a previous analysis of the loop. Loop invariants are always soundly reusable even if the loop has been modified, but they may introduce imprecision in the analysis. However, we provide a way to limit the loss of precision and show that in practice, only little imprecision are introduced. Both approaches are complementary and can be used together to accelerate a static analysis, either an ongoing one or an incremental one.

Both approaches are formalized and proved sound. They have been implemented in Eva [6], the abstract interpreter of Frama-C [3, 19], an analysis framework for C code. This implementation has been evaluated on a set of real-world commits of open-source programs: The evaluation shows that our incremental analysis is much more time-efficient than the original analysis when analyzing two successive commits, while remaining precise in practice. Therefore, we think that such an incremental analysis helps integrate sound static analyzers in CI/CD process.

Contributions

- An approach for summarizing functions and reusing the summaries in subsequent analyses;
- An approach for reusing loop invariants in subsequent analyses;
- A formalization and a proof of soundness for both approaches for an imperative language with dynamic allocations;
- An experimental evaluation on real-world commits of open-source C programs for both approaches.

The rest of the paper is organized as follows. Section 2 presents the background on abstract interpretation. Section 3 presents an example to illustrate our approach for reusing function summaries and loop invariants. Section 4 introduces a representative programming language and its formal semantics. Using this language, Sections 5 and 6 formalize our approaches for function summaries and loop invariants, respectively. Section 7 presents an implementation and evaluation of both approaches. Section 8 discusses related work. Section 9 concludes the paper and discusses future work.

2 Elements of Abstract Interpretation

This section introduces the basic elements of abstract interpretation that are used in the rest of the paper. Abstract interpretation [10, 9] is a general theory of sound approximation of program semantics. It is based on the idea of computing an over-approximation of the set of possible states of a program using abstract domains, and provides a formal guarantee that the computed over-approximation is sound, i.e., it contains all possible states of the program. More formally, starting from a so-called concrete semantic domain $\mathcal{D}$, which can express the properties of interest (such as the set of memory states, to prove non-reachability of erroneous states), but is too difficult to compute with, abstract interpretation introduces the concept of abstract domain $\mathcal{D}^\sharp$ where the elements of $\mathcal{D}^\sharp$ are sound approximations of the elements of $\mathcal{D}$. The domains $\mathcal{D}$ and $\mathcal{D}^\sharp$ are partially ordered by a relation $\sqsubseteq$ and $\sqsubseteq^\sharp$. We also define the concretization function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ as a monotonic function that maps an abstract element to a concrete element. We require that the abstract domain provides a sound approximation of the concrete join $\sqcup$ and meet $\sqcap$ operators, such that for all $x^\sharp, y^\sharp$ in $\mathcal{D}^\sharp$, $\gamma(x^\sharp) \sqcup \gamma(y^\sharp) \sqsubseteq \gamma(x^\sharp \sqcup^\sharp y^\sharp)$ and $\gamma(x^\sharp) \sqcap \gamma(y^\sharp) \sqsubseteq \gamma(x^\sharp \sqcap^\sharp y^\sharp)$. Moreover, the least upper bound $\top^\sharp$ (resp. greatest lower bound $\bot^\sharp$) of a set of abstract elements, when it exists, approximates the least upper bound $\top$ (resp. greatest lower bound $\bot$) of the corresponding set of concrete elements. A semantic operator is a function over domains. Abstract interpretation often requires computing fixpoints of semantic functions, e.g. in case of loops or recursive functions. They can be computed by iterating an abstract operator over an abstract domain. The convergence of the iteration sequence in the abstract domain is guaranteed by the use of a widening operator $\nabla^\sharp$ [10]. Formally, if $f : \mathcal{D} \mapsto \mathcal{D}$ is a monotonic operator over a complete concrete lattice $(\mathcal{D}, \sqsubseteq)$ and $f^\sharp : \mathcal{D}^\sharp \mapsto \mathcal{D}^\sharp$ is a sound abstraction of f, then the limit of the iteration sequence $x_0^\sharp = \bot^\sharp$, $x_{n+1}^\sharp = x_n^\sharp \ \nabla^\sharp \ f^\sharp(x_n^\sharp)$ converges in finite time to a stable iterate $x^\sharp$, which is a sound approximation of the least fixpoint lfp $f \sqsubseteq \gamma(x^\sharp)$. The existence of lfp f and the soundness of the approximation are guaranteed by Tarski's theorem [11, 34]. Moreover, the abstract operator $f^\sharp$ is not required to be monotonic, and does not necessarily admit a least fixpoint. We can settle for an abstract post-fixpoint $f^\sharp(x^\sharp) \sqsubseteq^\sharp x^\sharp$. Moreover, any post-fixpoint is an inductive invariant and the least fixpoint is the least invariant, so the limit of the iteration sequence is also an invariant. Figure 1 presents an interval abstract domain $\mathcal{I}_\mathbb{Z}^\sharp$, which represents values of integers as a pair of bounds, and a widening operator for fast and sound convergence of the iteration sequence.

3 Motivating Example

This section introduces an example that illustrates our contributions. Consider the following (buggy) C program that stores squares' size in an array before computing their area.

$$\mathcal{I}_{\mathbb{Z}}^{\sharp} \triangleq \{\ [a,b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\ \} \cup \{\bot_{\mathcal{I}_{\mathbb{Z}}}^{\sharp}\}$$

$$[a,b]\ \nabla_{\mathcal{I}_{\mathbb{Z}}}^{\sharp}\ [c,d] \triangleq \left[\begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases} \right]$$

Figure 1 Interval abstract domain and associated widening operator.

```
1  #define LEN 100
2  int area(int n) { return n + n; /* bug: should be 'n * n' */ }
3  void main() {
4    int sizes[LEN], areas[LEN];
5    for(int i = 0; i < LEN; i++)
6      sizes[i] = i;
7    for(int i = 0; i <= LEN; i++)  /* bug: should be 'i < LEN' */
8      areas[i] = area(sizes[i/2]);
9  }
```

This program contains one functional bug at line 2 and one undefined behavior at line 8 because of an incorrect test at line 7. Any modern abstract interpreter handling C code, such as Astrée [23], the Eva [4, 6] plugin of Frama-C [3, 19], Mopsa [24], or Polyspace [21], is able to analyze such a code instantaneously and reports one alarm. This alarm is raised when writing into the array `areas[100]`. Indeed, the loop at line 7 iterates one time too many, which results in an out-of-bound access. Such modern analyzers often implement function summaries that allow them to avoid analyzing twice a function under the same analysis context. An analysis context of a function is an over-approximation of the memory state at the call site, including the values of the arguments.

In our example, running an interval analysis with a standard widening operator allows the analyzer to converge in four iterations and to only analyze the function `area` twice instead of four times (i.e., one for each loop iteration). Indeed, for each loop iteration, the analysis context of the function call at line 8, in which `sizes` is omitted because it does not change throughout the analysis, is: $\{i \mapsto [0,0]; areas[i] \mapsto uninit; n \mapsto [0,0]\}$ at iteration 1, $\{i \mapsto [0,1]; areas[i] \mapsto [0,0]; n \mapsto [0,0]\}$ at iteration 2, $\{i \mapsto [0,100]; areas[i] \mapsto [0,100]; n \mapsto [0,50]\}$ at iteration 3 (with widening), and $\{i \mapsto [0,100]; areas[i] \mapsto [0,100]; n \mapsto [0,50]\}$ at iteration 4. We observe that the analysis of the function `area` depends on the value of its argument n (which is given by the value of array `sizes` at index `i/2`), and not on the value of the array `areas`. By assuming a sound over-approximation of the variables needed to analyze the function `area`, an analyzer can build a function summary that restricts the values of the state of the analysis before the call to `area` to the values of the variables needed to analyze the function. Such a state is then used to avoid the analysis of the body of the function again if the values of the needed variables are the same. The function summary in this case contains the state of the analysis before the call (denoted as input state $I^{\sharp}$) to function `area`, the state of the analysis after the call (denoted as output state $\mathcal{O}^{\sharp}$) to `area`, and the set (or sets) of variables needed to analyze the function. For example, the function summary for the call at line 8 after the first loop iteration is $\{I^{\sharp} = \{n \mapsto [0,0]\}; \mathcal{O}^{\sharp} = \{areas[i] \mapsto [0,0]\}; R^{\sharp} = \{n\}; W^{\sharp} = \{areas[i]\}\}$, where $R^{\sharp}$ (resp. $W^{\sharp}$) is the set of variables read (resp. written) by the function `area`. When analyzing the same function again during the second loop iteration, the value of the read variable n does not change, so the analyzer can reuse the function summary to avoid

reanalyzing the body of function `area`: It directly computes the state of the analysis after the call to function `area` from the output state $\mathcal{O}^\sharp$ and the written variables $W^\sharp$. Our first contribution consists in formalizing such function summaries for soundly reusing them when possible in order to speed up the analysis.

If such an analyzer is part of a CI/CD process, the developers would easily detect the runtime error at line 7 and fix it by submitting the following patch:

```
- for (i = 0; i <= LEN; i++)
+ for (i = 0; i < LEN; i++)
```

Analyzing the code again will allow them to prove that there is no more out-of-bound access in the program. However, for large programs, this analysis can be time-consuming, which may prevent its use in a CI/CD process. Indeed, it would require to run a new analysis from scratch for every program change, even minimal ones. This paper explains how to reuse previously computed function summaries from a previous analysis in order to prevent abstract interpreters from reanalyzing the body of the function `area` when reanalyzing the (almost) same program after a small patch. In fact, the analysis of the call to `area` was already done prior to the patch for the call contexts $\{n \mapsto [0,0]\}$ and $\{n \mapsto [0,50]\}$. The context $\{n \mapsto [0,0]\}$ can be directly reused when analyzing the patched version, so the analyzer only needs to analyze the call for the context $\{n \mapsto [0,49]\}$.

Later, the functional bug at line 2 will also be fixed by the following commit:

```
- int area(int n) { return n + n; }
+ int area(int n) { return n * n; }
```

Modifying the function `area` results in invalidating all function summaries inferred for this function and for all functions that call it. Therefore, the analyzer will analyze again from scratch the functions `area` and `main`.

Our second contribution consists in reusing previously computed loop invariants to speed up the analysis of both loops in our running example in order to speed up the analysis even when no function summaries can be soundly reused. Indeed, prior to the patch for the functional bug, the analyzer has already computed the loop invariants for the loops at lines 5 and 7. If $arr[a,b] \mapsto [i,j]$ denotes *the array arr contains values in the interval* $[i,j]$ *for any index in the interval* $[a,b]$, these invariants are $\{sizes[0,99] \mapsto [0,99]; areas[0,99] \mapsto uninit; i \mapsto [0,99]\}$ and $\{sizes[0,99] \mapsto [0,99]; areas[0,99] \mapsto [0,198]; i \mapsto [0,99]\}$, respectively. Reusing them will allow the analyzer to converge faster by starting the iteration from the previously computed fixpoint for this loop instead of the least element $\bot^\sharp$. This optimization allows the first loop to converge in one iteration only as the fixpoint has not changed, while the second loop will converge in two iterations as the value of $areas[0,49]$ still needs to stabilize to $[0,2401]$ ($2401 = 49 \times 49$).

Together, our contributions allow analyzers to reduce the analysis time significantly when submitting small patches that fix bugs or introduce new features on large programs (see Section 7). Therefore, they help integrate such analyzers in a CI/CD process.

4 Language and Semantics

Figure 2 presents our language. It is an imperative language containing basic control flows (sequences, conditionals and loops), assignments, function calls and dynamic allocations, which allocate new memory locations of type integer at a given allocation site μ. Expressions consist of integer constants, variables, dereferences, addresses of a variable, and basic n-ary

$$
\begin{array}{lll}
\textbf{expr} & e ::= c & c \in \mathbb{V} \\
& \mid *e & \text{(dereference)} \\
& \mid \&x & \text{(address of } x\text{)},\ x \in \mathbb{X} \\
& \mid \diamondsuit(e_1, \ldots, e_n) & \diamondsuit \in \{+, -, *, /\} \\
\textbf{stmt} & s ::= *e_1 \coloneqq e_2 & \text{(assignment)} \\
& \mid *(\&x) \coloneqq \text{malloc}_\mu & \text{(allocation)},\ \mu \in \mathbb{L} \\
& \mid s_\alpha ; s_\beta & \text{(sequence)} \\
& \mid f(\vec{e}) & \text{(function call)} \\
& \mid \textbf{if } (e \bowtie 0) \textbf{ then } s_\alpha \textbf{ else } s_\beta & \text{(conditional)} \\
& \mid \textbf{while } (e \bowtie 0) \textbf{ do } s & \text{(loop)} \\
\textbf{fun} & f ::= f(\vec{x})\{s\} & \text{(function definition)} \\
& \bowtie \in \{=, \neq, <, >, \leq, \geq\} & \text{(relational operator)}
\end{array}
$$

Figure 2 Syntax of our language.

arithmetic operators over integers. Functions are defined with a unique name f, a list of formal parameters $\vec{x}$, and a body of statements s. Variables denote integers or memory locations.

Notations

We denote $\mathbb{X}$, the finite set of all variables and formal parameters of functions; $\mathbb{L}$, the finite set of allocation sites; $\mathbb{H} = \mathbb{L} \times \mathbb{N}$, pairs of allocation site and the number of allocations at this site that represents a memory block dynamically allocated in the heap; $\mathbb{A} = \mathbb{X} \uplus \mathbb{H}$, the set of all possible memory locations; $\mathbb{V} = \mathbb{Z} \uplus \mathbb{A}$, the set of all possible values (either an integer or a memory location) assignable to a variable; and $\mathbb{M} : \mathbb{A} \rightharpoonup \mathbb{V}$, a memory state that maps a memory location to some (concrete) value. $\mapsto$ is the update operator that assigns a value to a memory location in a memory state, such that $m[a \mapsto v]$ denotes the memory state $m \in \mathbb{M}$ that maps the location $a \in \mathbb{A}$ to the value $v \in \mathbb{V}$ and other elements $a' \in \text{dom}(m)$ to their previous value $m(a')$. $|$ is the restriction operator that removes the keys of a mapping, such that $m|_A$ is the memory state m restricted to the set A of memory locations. Additionally, we denote $e^\sharp$ an element of an abstract domain. Furthermore, the formal parameters of a function are given by params : **fun** $\to \mathcal{P}(\mathbb{X})$, the body of the function by body : **fun** $\to$ **stmt** and the local variables of the function by locals : **fun** $\to \mathcal{P}(\mathbb{X})$.

Assumptions and limitations

We do not handle pointer arithmetic: $[\![\diamondsuit]\!]$ is only defined for integers. Assignments can only write a memory location by dereferencing it. So, assigning some expression e to a variable $x \in \mathbb{X}$ is written $*(\&x) = e$. Every malloc_μ expression is assumed to be unique and $\mu \in \mathbb{L}$ identifies the allocation site. A same malloc_μ can be called multiple times during the execution of a program (typically, when written in a loop). In such a case, it generates multiple memory blocks at different addresses. Without loss of generality, all variable names are assumed to be unique, including formal parameters of defined functions. Functions have no return value: They can be simulated by passing to the function a reference to a variable

in which the function stores its result. Recursive function calls are also not supported, we already have functions, unbounded loops and unbounded memory allocation, so the support of recursive calls does not bring theoretical difficulties. It is just a technical limitation of the formalization and is left for future work.

4.1 Concrete Semantics

Figures 3 and 4 define the concrete semantic functions for expressions and statements, respectively. The semantics for expressions, $\mathbb{E}[\![\mathbf{expr}]\!]$, maps a concrete memory state to a concrete value. The semantics for statements, $\mathbb{S}[\![\mathbf{stmt}]\!]$, maps a concrete memory state to another concrete memory state: $\mathbb{S}[\![\mathbf{stmt}]\!]$ computes the state of the program reached at the end of the execution of a statement. The concrete semantics is deterministic and $\mathbb{S}[\![\mathbf{stmt}]\!]$ is a partial function to handle undefined operations such as divisions by zero or reading uninitialized memory locations. A function call is done by binding the value of the evaluation of the arguments to the formal parameters in the memory state, executing the body of the function, removing the formal parameters from the memory state at the end of the function execution, and returning this memory state. The semantics for dynamic allocations relies on a next operator: for a location site $\mu \in \mathbb{L}$ and a memory state $m \in \mathbb{M}$, $\text{next}(\mu, m)$ returns the first free memory block in $\mathbb{H}$ at this call site, which is formally defined as follows:

$$\text{next}(\mu, m) \triangleq \langle \mu, \min\{i \in \mathbb{N} \mid \langle \mu, i\rangle \notin \text{dom}(m)\}\rangle.$$

This definition and the concrete semantics for dynamic allocations imply that the first memory block allocated at μ is $\langle \mu, 0\rangle$, the second one is $\langle \mu, 1\rangle$, and so on.

$$\begin{aligned}
\mathbb{E}[\![\mathbf{expr}]\!] &: \mathbb{M} \to \mathbb{V} \\
\mathbb{E}[\![c]\!]m &\triangleq c,\ c \in \mathbb{Z} \\
\mathbb{E}[\![*e]\!]m &\triangleq m(\mathbb{E}[\![e]\!]m) \\
\mathbb{E}[\![\&x]\!]m &\triangleq x,\ x \in \mathbb{X} \\
\mathbb{E}[\![\diamondsuit(e_1, ..., e_n)]\!]m &\triangleq \diamondsuit(\mathbb{E}[\![e_1]\!]m, ..., \mathbb{E}[\![e_2]\!]m)
\end{aligned}$$

Figure 3 Concrete semantics for expressions.

4.2 Abstract Domains

Value Abstract Domain

We abstract the concrete values $\mathbb{V}$ by an abstract domain $\mathbb{V}^\sharp$, such that $(\mathbb{V}^\sharp, \sqsubseteq^\sharp_{\mathbb{V}})$ is a partial order abstracting the concrete domain $(\mathcal{P}(\mathbb{V}), \subseteq)$ featuring a monotonic concretization function $\gamma_{\mathbb{V}} : \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{V})$ and a widening operator $\nabla^\sharp_{\mathbb{V}}$. To construct such an abstract value domain, we need to abstract both integer values and memory locations.

The abstract domain $(\mathbb{Z}^\sharp, \sqsubseteq^\sharp_{\mathbb{Z}})$ for integer values is left unspecified. We just assume a concretization function $\gamma_{\mathbb{Z}} : \mathbb{Z}^\sharp \to \mathcal{P}(\mathbb{Z})$ and a widening operator $\nabla^\sharp_{\mathbb{Z}}$. For instance, this abstract domain is the interval domain in our running example. The set of variables is finite, so variable sets can be abstracted by its powerset $(\mathcal{P}(\mathbb{X}), \subseteq)$. An abstract heap address represents a set of concrete heap addresses [17]. Consequently, memory updates on abstract heap addresses are done by merging the new value with the previous value, which is called a

$$
\begin{aligned}
\mathbb{S}[\![\mathbf{stmt}]\!] &: \mathbb{M} \rightharpoonup \mathbb{M} \\
\mathbb{S}[\![*e_1 := e_2]\!]m &\triangleq m[\mathbb{E}[\![e_1]\!]m \mapsto \mathbb{E}[\![e_2]\!]m] \\
\mathbb{S}[\![*(\&x) := \mathrm{malloc}_\mu]\!]m &\triangleq \text{let } \langle \mu, n\rangle := \mathrm{next}(\mu, m) \text{ in} \\
&\quad\; m[\mathbb{E}[\![\&x]\!]m \mapsto \langle \mu, n\rangle][\langle \mu, n\rangle \mapsto 0] \\
\mathbb{S}[\![f(\vec{e})]\!]m &\triangleq \text{let } \vec{x} := \mathrm{params}(f) \text{ in} \\
&\quad\; \text{let } \vec{y} := \mathrm{locals}(f) \text{ in} \\
&\quad\; \text{let } m_{\mathrm{ret}} := \mathbb{S}[\![\mathrm{body}(f)]\!](m[\vec{x} \mapsto \mathbb{E}[\![\vec{e}]\!]m]) \text{ in} \\
&\quad\; m_{\mathrm{ret}}|_{\mathrm{dom}(m_{\mathrm{ret}}) \setminus (\vec{x} \cup \vec{y})} \\
\mathbb{S}[\![s_\alpha; s_\beta]\!]m &\triangleq (\mathbb{S}[\![s_\beta]\!] \circ \mathbb{S}[\![s_\alpha]\!])m \\
\mathbb{S}[\![\mathbf{if}\ (e \bowtie 0)\ \mathbf{then}\ s_\alpha\ \mathbf{else}\ s_\beta]\!]m &\triangleq \mathbf{if}\ (\mathbb{E}[\![e]\!]m \bowtie 0)\ \mathbf{then}\ \mathbb{S}[\![s_\alpha]\!]m\ \mathbf{else}\ \mathbb{S}[\![s_\beta]\!]m \\
\mathbb{S}[\![\mathbf{while}\ (e \bowtie 0)\ \mathbf{do}\ s]\!]m &\triangleq \mathrm{lfp}(F)(m) \\
&\quad \text{where } F(f)(m) \triangleq \begin{cases} f(\mathbb{S}[\![s]\!]m) & \text{if } \mathbb{E}[\![e]\!]m \bowtie 0 \\ m & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4 Concrete semantics for statements.

weak update. The induced precision loss with weak updates can be compensated by using more sophisticated techniques to summarize heap addresses [18]. The idea is to allow some abstract heap addresses to represent a singleton concrete heap address and allow *strong updates* on such addresses. We choose to represent abstract heap addresses $\mathbb{H}^\sharp_k : \mathbb{L} \times \mathbb{N}_{\leq k}$ as a pair of allocation site $\mu \in \mathbb{L}$ and a natural number smaller or equal than k.

Such a technique is based on *k-limiting* [20] and *allocation site* [2]. The natural number k is used to limit the maximum number of generation of strong abstract heap addresses on a given site before over-approximating further allocations with a weak abstract heap address. Such limit k will be referred as $\mathrm{k_{lim}}$ or k for subscript in the rest of the paper. More precisely, we want to represent the first k allocations on a site μ with a dedicated abstract heap address $\langle \mu, k\rangle$ for all $0 \leq k < \mathrm{k_{lim}}$, and all further allocations with an abstract heap address $\langle \mu, \mathrm{k_{lim}}\rangle$. Consequently, when an abstract heap address represents a singleton concrete heap address, the update on this address is a strong update: the new value replaces the previous one. Otherwise, if an abstract heap address potentially represents multiple concrete heap addresses, the update is a weak update: the new value is merged with the previous one, which is the case for $\langle \mu, \mathrm{k_{lim}}\rangle$. Formally, the abstract value domain $\mathbb{V}^\sharp$ is defined as follows:

$$
\begin{aligned}
\mathbb{V}^\sharp &\triangleq \mathbb{Z}^\sharp \times \mathcal{P}(\mathbb{H}^\sharp_k) \times \mathcal{P}(\mathbb{X}) \cup \{\top^\sharp_{\mathbb{V}}, \bot^\sharp_{\mathbb{V}}\} \\
\gamma_{\mathbb{V}} &: \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{V}) \\
\gamma_{\mathbb{V}}(\top^\sharp_{\mathbb{V}}) &\triangleq \mathbb{V} \\
\gamma_{\mathbb{V}}(\bot^\sharp_{\mathbb{V}}) &\triangleq \emptyset \\
\gamma_{\mathbb{V}}(z^\sharp, h^\sharp, x^\sharp) &\triangleq \gamma_{\mathbb{Z}}(z^\sharp) \cup \gamma_{\mathbb{H}}(h^\sharp) \cup x^\sharp \\
&\quad \text{where } \gamma_{\mathbb{H}}(h^\sharp) \triangleq \bigcup \{\gamma_h(\langle \mu, n\rangle) \mid \forall \langle \mu, n\rangle \in h^\sharp\} \\
&\quad \text{where } \gamma_h(\langle \mu, n\rangle) \triangleq \begin{cases} \{\langle \mu, n\rangle\} & \text{if } n < k \\ \{\langle \mu, j\rangle \mid \forall j \geq k\} & \text{if } n = k. \end{cases}
\end{aligned}
$$

The definitions of operators $\sqsubseteq^\sharp_{\mathbb{V}}$, $\sqcup^\sharp$, $\sqcap^\sharp$ and $\nabla^\sharp$ are omitted, but defined pointwise and extended to $\top^\sharp_{\mathbb{V}}$ and $\bot^\sharp_{\mathbb{V}}$ such that the former is the greatest element of $\mathbb{V}^\sharp$ and the latter is its least element for the partial order $\sqsubseteq^\sharp$.

Non-relational Memory Abstract Domain

We derive the abstraction $\mathbb{M}^\sharp$ of $\mathbb{M} : \mathbb{A} \rightharpoonup \mathbb{V}$ from the abstract value domain $\mathbb{V}^\sharp$ by mapping each memory location in the finite set of abstract memory locations $\mathbb{A}^\sharp_k : \mathbb{X} \uplus \mathbb{H}^\sharp_k$ to an abstract value $\mathbb{V}^\sharp$. The abstract memory domain $\mathbb{M}^\sharp$ is thus a partial mapping from abstract memory locations to abstract values. The operations $\sqsubseteq^\sharp, \nabla^\sharp, \sqcup^\sharp, \sqcap^\sharp$ are defined pointwise, and the least element $\bot^\sharp_{\mathbb{V}}$ is used when one of the operands is undefined . The least element $\bot^\sharp$ (resp. the greatest element $\top^\sharp$) maps each memory location in $\mathrm{dom}(\mathbb{M}^\sharp)$ to the least element $\bot^\sharp_{\mathbb{V}}$ (resp. the greatest element $\top^\sharp_{\mathbb{V}}$) of the abstract value domain. The concretization function γ takes into account possible undefined memory locations. Formally, the abstract memory domain is defined as follows:

$$\mathbb{M}^\sharp \triangleq (\mathbb{A}^\sharp_k \rightharpoonup \mathbb{V}^\sharp)$$
$$\gamma : \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{M})$$
$$\gamma(M^\sharp) \triangleq \left\{ m \in \mathbb{M} : \left\{ \begin{array}{ll} m(\langle a, n\rangle) \in \gamma_{\mathbb{V}}(M^\sharp(\langle a, n\rangle)) & (1) \\ m(\langle a, n\rangle) \in \gamma_{\mathbb{V}}(M^\sharp(\langle a, k\rangle)) & (2) \\ m(x) \in \gamma_{\mathbb{V}}(M^\sharp(x)) & (3) \end{array} \right. \middle| \; \forall a : j_a \in \mathbb{N} \right\}$$

where (1) $\forall \langle a, n\rangle \in \mathrm{dom}(M^\sharp)$ and $n < k$ and $n < j_a$
(2) if $\langle a, k\rangle \in \mathrm{dom}(M^\sharp)$ and $k \leq n \leq j_a$
(3) $\forall x \in \mathbb{X}$ and $x \in \mathrm{dom}(M^\sharp)$

4.3 Abstract Semantics

Figures 5 and 6 present the abstract semantics of our language, which computes an over-approximation of the concrete semantics. The abstract semantics for expressions, $\mathbb{E}^\sharp[\![\mathbf{expr}]\!]$, maps an abstract memory to an abstract value, while the abstract semantics for statements, $\mathbb{S}^\sharp[\![\mathbf{stmt}]\!]$, maps an abstract memory to another abstract memory. $\mathbb{S}^\sharp[\![\mathbf{stmt}]\!]$ computes an over-approximation of the possible states reached at the end of the executed statement. Function $\phi_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z}^\sharp$ (resp. $\diamondsuit^\sharp$) is assumed to be a sound abstraction of integer constants (resp. unary operators) given by the abstract domain $\mathbb{Z}^\sharp$ for integer values. We assume given a sound abstraction of relational operators $\bowtie^\sharp_e$: $\mathbb{M}^\sharp \to \mathbb{M}^\sharp$ and its negation $\not\bowtie^\sharp_e : \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ such that for all non-relational abstract elements $M^\sharp \in \mathbb{M}^\sharp$ and all concrete memory states $m \in \mathbb{M}$, if $\mathbb{E}[\![e]\!]m \bowtie 0$ then $m \in \gamma(\bowtie^\sharp_e (M^\sharp))$, and its negation if $\mathbb{E}[\![e]\!]m \not\bowtie 0$ then $m \in \gamma(\not\bowtie^\sharp_e(M^\sharp))$.

The abstract semantics of an assignment performs an abstract memory update, which is either a strong update when the written address is precise, or a weak update when it is not. The formal definition of such a memory update is as follows:

▶ **Definition 1** (Memory Update). *Given an abstract memory $M^\sharp \in \mathbb{M}^\sharp$, a set of abstract memory locations $X \in \mathcal{P}(\mathbb{A}^\sharp_k)$ and an abstract value $v^\sharp \in \mathbb{V}^\sharp$, the update of $M^\sharp$ with X is defined as follows:*

$$\forall X \in \mathcal{P}(\mathbb{A}^\sharp_k), \forall v^\sharp \in \mathbb{V}^\sharp :$$
$$M^\sharp[X \rightsquigarrow v^\sharp] \triangleq \begin{cases} M^\sharp[a^\sharp \mapsto v^\sharp] & \text{if } X = \{a^\sharp\} \wedge \\ & (a^\sharp \in \mathbb{X} \vee (a^\sharp = \langle \mu, n\rangle \wedge n < k)) \\ & \text{strong update} \\ M^\sharp[\forall a^\sharp \in X. a^\sharp \mapsto v^\sharp \sqcup^\sharp_{\mathbb{V}} M^\sharp(a^\sharp)] & \text{otherwise weak update} \end{cases}$$

$$
\begin{aligned}
\mathbb{E}^\sharp[\![\textbf{expr}]\!] &: \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
\mathbb{E}^\sharp[\![c]\!]M^\sharp &\triangleq \phi_{\mathbb{Z}}(c) \\
\mathbb{E}^\sharp[\![*e]\!]M^\sharp &\triangleq \bigsqcup_{\mathbb{V}}^{\sharp}\{M^\sharp(a^\sharp) \mid \forall a^\sharp \in \mathbb{E}^\sharp[\![e]\!]M^\sharp\} \\
\mathbb{E}^\sharp[\![\&x]\!]M^\sharp &\triangleq \{x\} \\
\mathbb{E}^\sharp[\![\Diamond(e_1, ..., e_n)]\!]M^\sharp &\triangleq \Diamond^\sharp(\mathbb{E}^\sharp[\![e_1]\!]M^\sharp, \dots, \mathbb{E}^\sharp[\![e_n]\!]M^\sharp)
\end{aligned}
$$

Figure 5 Abstract semantics for expressions.

$$
\begin{aligned}
\mathbb{S}^\sharp[\![\textbf{stmt}]\!] &: \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
\mathbb{S}^\sharp[\![*e_1 := e_2]\!]M^\sharp &\triangleq M^\sharp[\mathbb{E}^\sharp[\![e_1]\!]M^\sharp \rightsquigarrow \mathbb{E}^\sharp[\![e_2]\!]M^\sharp] \\
\mathbb{S}^\sharp[\![*(\&x) := \text{malloc}_\mu]\!]M^\sharp &\triangleq \text{let } \langle\mu, n\rangle := \text{next}^\sharp(\mu, M^\sharp) \text{ in} \\
&\quad\; M^\sharp[\{x\} \mapsto \langle\mu, n\rangle][\langle\mu, n\rangle \rightsquigarrow Z_0^\sharp] \\
\mathbb{S}^\sharp[\![f(\vec{e})]\!]M^\sharp &\triangleq \text{let } \vec{x} := \text{params}(f) \text{ in} \\
&\quad\; \text{let } \vec{y} := \text{locals}(f) \text{ in} \\
&\quad\; \text{let } M^\sharp_{\text{ret}} = \mathbb{S}^\sharp[\![\text{body}(f)]\!]M^\sharp[\vec{x} \mapsto \mathbb{E}^\sharp[\![\vec{e}]\!]M^\sharp] \text{ in} \\
&\quad\; M^\sharp_{\text{ret}}|_{\text{dom}(M^\sharp_{\text{ret}})\setminus(\vec{x}\cup\vec{y})} \\
\mathbb{S}^\sharp[\![\textbf{if } (e \bowtie 0) \textbf{ then } s_\alpha \textbf{ else } s_\beta]\!]M^\sharp &\triangleq (\mathbb{S}^\sharp[\![s_\alpha]\!] \circ \bowtie^\sharp_e)\mathbb{M}^\sharp \sqcup^\sharp (\mathbb{S}^\sharp[\![s_\beta]\!] \circ \not\bowtie^\sharp_e)\mathbb{M}^\sharp \\
\mathbb{S}^\sharp[\![\textbf{while } (e \bowtie 0) \textbf{ do } s]\!]M^\sharp &\triangleq \not\bowtie^\sharp_e(\lim F^\sharp) \\
&\quad \text{where } F^\sharp(I^\sharp) \triangleq I^\sharp \;\nabla^\sharp\; (M^\sharp \sqcup^\sharp (\mathbb{S}^\sharp[\![s]\!] \circ \bowtie^\sharp_e)I^\sharp)
\end{aligned}
$$

Figure 6 Abstract semantics for statements.

The abstract semantics of a dynamic allocation relies on a $\text{next}^\sharp$ operator, which is an abstraction of the next operator. It bounds the number of allocations made on a site to k. The default value of an allocated memory location is denoted as $Z_0^\sharp$ in $\mathbb{Z}^\sharp$, and is a sound abstraction of the concrete initial allocation value 0. Formally:

▶ **Definition 2** ($\text{next}^\sharp$ operator)**.** *Given an abstract memory $M^\sharp \in \mathbb{M}^\sharp$, an allocation site $\mu \in \mathbb{L}$ and k_{lim}, the $next^\sharp$ function is defined as follows:*

$$
next^\sharp(\mu, M^\sharp) \triangleq \begin{cases} \langle\mu, k_{lim}\rangle & \textit{if } \forall i \in \mathbb{N}_{\leq k_{lim}}, \langle\mu, i\rangle \in dom(M^\sharp) \\ \langle\mu, n\rangle & \textit{where } n = \min\{i \in \mathbb{N} \mid \langle\mu, i\rangle \notin dom(M^\sharp)\} \quad \textit{otherwise} \end{cases}
$$

In a usual way, the abstract semantics for a loop relies on fixpoints, which is the limit of the iterates of a function $F^\sharp$. This limit is defined as follows.

▶ **Definition 3** (Limit of iterates)**.** *$lim\ F^\sharp$ computes the limit of the iterates the function $F^\sharp$ (defined in Figure 6), such that:*

$$
lim\ F^\sharp \triangleq F^{\sharp\delta}(\bot^\sharp) \quad \textit{where } \delta \textit{ is the minimal value such that } F^{\sharp\delta+1}(\bot^\sharp) = F^{\sharp\delta}(\bot^\sharp).
$$

Soundness Requirements of Abstract Semantic Functions

The concrete semantic functions are implicitly lifted to sets. We require that the abstract semantic functions are sound abstractions of the corresponding concrete semantic functions. This is expressed by the following soundness criterion: $\forall M^\sharp \in \mathbb{M}^\sharp$, $\mathbb{S}[\![\mathbf{stmt}]\!]\gamma(M^\sharp) \subseteq \gamma(\mathbb{S}^\sharp[\![\mathbf{stmt}]\!]M^\sharp)$ and $\mathbb{E}[\![\mathbf{expr}]\!]\gamma(M^\sharp) \subseteq \gamma_{\mathbb{V}}(\mathbb{E}^\sharp[\![\mathbf{expr}]\!]M^\sharp)$.

Set of Abstract Read, Written and Allocated Memory Locations

We define the set of abstract memory locations used in expressions, read in statements, written in statements, and allocated in statements. These sets are used in the definition of function summaries and loop invariants. They are defined inductively over the semantics of statements. These locations are collected and propagated throughout each evaluation of an expression and a statement. We only present here the important cases of these definitions. The others are defined in a straightforward manner.

▶ **Definition 4** (Set of abstract memory locations used in expressions). *The set of abstract memory locations used in an expression* ***expr*** *is the set of memory locations which might change the result of the abstract evaluation of the expression.*

$$\begin{aligned}
\Upsilon^\sharp[\![\boldsymbol{expr}]\!] &: \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{A}_k^\sharp) \\
\Upsilon^\sharp[\![\&x]\!]M^\sharp &\triangleq \emptyset \\
\Upsilon^\sharp[\![*e]\!]M^\sharp &\triangleq \mathit{let}\ Y := \Upsilon^\sharp[\![e]\!]M^\sharp \cup \mathbb{E}^\sharp[\![e]\!]M^\sharp\ \mathit{in} \\
&\qquad \mathit{let}\ X := \mathbb{E}^\sharp[\![*e]\!]M^\sharp\ \mathit{in} \\
&\qquad\quad \begin{cases} Y & \mathit{if}\ |X| = 1 \\ & \wedge\ (\forall a^\sharp \in X : a^\sharp \in \mathbb{X} \vee (a^\sharp = \langle \mu, n\rangle \wedge n < k)) \\ X \cup Y & \mathit{otherwise} \end{cases} \\
\Upsilon^\sharp[\![\diamondsuit(e_1, ..., e_n)]\!]M^\sharp &\triangleq \bigcup_{\forall i \in [1,n]} \Upsilon^\sharp[\![e_i]\!]M^\sharp
\end{aligned}$$

▶ **Definition 5** (Set of abstract read memory locations in statements). *An abstract memory location $a^\sharp$ is read by a statement* ***stmt*** *if its abstract value $M^\sharp(a^\sharp)$ is accessed by the computation of $\mathbb{S}^\sharp[\![\boldsymbol{stmt}]\!]M^\sharp$.*

$$\begin{aligned}
\mathit{Read}^\sharp[\![\boldsymbol{stmt}]\!] &: \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{A}_k^\sharp) \\
\mathit{Read}^\sharp[\![*e_1 := e_2]\!]M^\sharp &\triangleq \Upsilon^\sharp[\![e_1]\!]M^\sharp \cup \Upsilon^\sharp[\![e_2]\!]M^\sharp \\
\mathit{Read}^\sharp[\![*(\&x) := \mathit{malloc}_\mu]\!]M^\sharp &\triangleq \emptyset \\
\mathit{Read}^\sharp[\![f(\vec{e})]\!]M^\sharp &\triangleq \mathit{let}\ M_1^\sharp := M^\sharp[\mathit{params}(f) \mapsto \mathbb{E}^\sharp[\![\vec{e}]\!]M^\sharp]\ \mathit{in} \\
&\qquad \Upsilon^\sharp[\![\vec{e}]\!]M^\sharp \cup \mathit{Read}^\sharp[\![\mathit{body}(f)]\!]M_1^\sharp \setminus \mathit{locals}(f) \setminus \mathit{params}(f)
\end{aligned}$$

▶ **Definition 6** (Set of abstract memory locations modified in statements). *An abstract memory location $a^\sharp$ is written by a statement* ***stmt*** *if it is a lvalue in an assignment statement.*

$$\begin{aligned}
\mathit{Write}^\sharp[\![\boldsymbol{stmt}]\!] &: \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{A}_k^\sharp) \\
\mathit{Write}^\sharp[\![*e_1 := e_2]\!]M^\sharp &\triangleq \mathbb{E}^\sharp[\![e_1]\!]M^\sharp \\
\mathit{Write}^\sharp[\![*(\&x) := \mathit{malloc}_\mu]\!]M^\sharp &\triangleq \{x\} \\
\mathit{Write}^\sharp[\![f(\vec{e})]\!]M^\sharp &\triangleq \mathit{let}\ M_1^\sharp := M^\sharp[\mathit{params}(f) \mapsto \mathbb{E}^\sharp[\![\vec{e}]\!]M^\sharp]\ \mathit{in} \\
&\qquad \mathit{Write}^\sharp[\![\mathit{body}(f)]\!]M_1^\sharp \setminus \mathit{params}(f) \setminus \mathit{locals}(f)
\end{aligned}$$

▶ **Definition 7** (Set of abstract allocated memory locations in statements). *An abstract memory location $a^\sharp$ is allocated by a statement* ***stmt*** *if it is the result of a call to the allocation function.*

$$Alloc^\sharp[\![\textbf{\textit{stmt}}]\!] : \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{A}_k^\sharp)$$
$$Alloc^\sharp[\![*(\&x) := malloc_\mu]\!]M^\sharp \triangleq \{next^\sharp(\mu, M^\sharp)\}$$
$$Alloc^\sharp[\![f(\vec{e})]\!]M^\sharp \triangleq Alloc^\sharp[\![body(f)]\!](M^\sharp[params(f) \mapsto \mathbb{E}^\sharp[\![\vec{e}]\!]M^\sharp])$$

▶ **Example 8.** Given a statement $x := \text{malloc}_\mu; *x := y$ (written in C-like for simplicity) and an abstract memory $M^\sharp$ such that $M^\sharp = \{\langle\mu, 0\rangle \mapsto 0, y \mapsto 1\}$, if $\text{k}_{\text{lim}} = 1$ we have:

$$\mathbb{S}^\sharp[\![.]\!]M^\sharp = M^\sharp[x \mapsto \langle\mu, 1\rangle, \langle\mu, 1\rangle \mapsto 1]$$
$$Read^\sharp[\![.]\!]M^\sharp = \{x, y, \langle\mu, 1\rangle\}, Write^\sharp[\![.]\!]M^\sharp = \{x, \langle\mu, 1\rangle\}, Alloc^\sharp[\![.]\!]M^\sharp = \{\langle\mu, 1\rangle\}$$

We have to consider $\langle\mu, 1\rangle$ as read because it is a weak memory location, thus its value is read during a weak update.

▶ **Definition 9** (Projection). *Given a set of memory locations $X^\sharp \in \mathcal{P}(\mathbb{A}_k^\sharp)$ and an abstract memory $M^\sharp \in \mathbb{M}^\sharp$, the projection of $M^\sharp$ on $X^\sharp$ is defined as:*

$$M^\sharp \downarrow X^\sharp \triangleq \lambda x.\textit{if } x \in X^\sharp \textit{ then } M^\sharp(x) \textit{ else } \top^\sharp$$

Memory locations are omitted from the abstract memory after a projection if their value is $\top^\sharp$ for the sake of readability.

5 Incremental Analysis with Function Summaries

This section introduces a summarization technique that allows abstract interpreters to avoid reanalyzing the body of a function when calling it within the same analysis context. We first introduce the program matching process that identifies the unmodified functions across two program versions. We then present and formalize the concept of function summaries and how they can be soundly reused to speed up incremental static analysis.

Given two program versions P_{new} and P_{old}, we aim to identify the functions that have not changed between the two versions. We consider two functions f_{new} and f_{old} to be similar if their ASTs have the same shape, they have the same instructions, the set of global variables they access are the same and have not been modified, and the functions they call are similar. Although simple, this matching process is sufficient for our purpose and allows us to identify a significant number of functions that have not changed between two versions of a program.

▶ **Example 10.** Consider the following program:

```
1 int a = 0;
2 void f() { a = 0; }
3 void g() { a = 1; }
4 void main() { f(); g(); }
```

Let us consider two different patches (a) and (b) that are applied independently to the program:

```
- int a = 0;
+ int a = 1;
```

(a)

```
- void f() { a = 0; }
+ void f() { a = 1; }
```

(b)

Applying patch (a) directly changes the value of the global variable a. Therefore, the functions `f`, `g` and their caller `main` are considered different from their previous versions. In contrast, applying patch (b) only changes the body of the function `f`, which impacts its caller `main` but not the function `g` that is independent of `f`. Therefore, the function `g` is considered similar to its previous version by the matching algorithm. Such a matching process is efficient (with regard to the number of functions that are considered similar) when the changes are localized, i.e., when they do not modify global variables accessed by many functions or when they do not change functions that are called by many other functions. More sophisticated techniques exist to identify similar functions or programs for more complex changes [13, 15], such as detecting variable renaming. However, these are beyond the scope of this work and are left for future research.

After identifying the functions that have not changed between two program versions, we aim to reuse the function summaries for these functions to speed up an incremental analysis. We now introduce the concept of function summaries and how they can be soundly reused to avoid reanalyzing the body of a function. In essence, a function summary is a pair of input/output abstract memory states $(\mathcal{I}^\sharp, \mathcal{O}^\sharp)$ that represent the analysis of a function's body. The input state $\mathcal{I}^\sharp$ is the abstract memory state before executing the body of the function and the output state $\mathcal{O}^\sharp$ is the abstract memory state after executing the body of the function. We consider $\mathcal{I}^\sharp$ as the analysis context of f that contains the abstract values of the function's arguments. By storing $(\mathcal{I}^\sharp, \mathcal{O}^\sharp)$ in a cache, we can reuse the output state $\mathcal{O}^\sharp$ when analyzing the same function body with an equivalent input state $\mathcal{I}^\sharp_{\text{new}}$, such as $\mathcal{I}^\sharp_{\text{new}} = \mathcal{I}^\sharp$. However, such a reusability condition is too restrictive and results in many cache misses because the input abstract memory state $\mathcal{I}^\sharp$ contains memory locations that are not necessarily needed to analyze function f. The following example illustrates this issue.

▶ **Example 11.** Consider the following program:

```
1 void f(int *p) { *p = 1; }
2 void main() {
3   int a = 0;
4   f(&a);
5 }
```

At line 4, the analysis infers $(\mathcal{I}^\sharp_4 = \{a \mapsto 0,\ p \mapsto \{\&a\}\}, \mathcal{O}^\sharp_4 = \{a \mapsto 1, p \mapsto \{\&a\}\})$ as the summary of `f`. A developer then applies the following patch that adds two more calls to `f`:

```
4   f(&a);
5 + f(&a);
6 + f(&a);
```

The incremental analysis of the program identifies that function `f` has not changed between both versions of the program, so it tries to reuse the summary of the first call to function `f` at line 4 when analyzing the second call to function `f` at line 5 and the third call to `f` at line 6. Indeed, the effect of the analysis of function `f` for every call is identical as the value of variable a does not impact the analysis of function `f`. However, the reusability condition $\mathcal{I}^\sharp_{\text{new}} = \mathcal{I}^\sharp$ does not hold at line 5 since $\{a \mapsto 1\} \neq \{a \mapsto 0\}$, the first mapping being part of the input state $\mathcal{I}^\sharp_5$ at line 5 and the second one being part of $\mathcal{I}^\sharp_4$. Consequently, the analyzer should analyze the body of `f` again and compute a new summary for the call at line 5, while it could be soundly reused in this case. Therefore, the above-mentioned reusability condition is too restrictive and leads to a lot of cache misses, although correct. Additionally, the abstract input and output states can be quite large for a real-world program, leading to significant

memory overhead. In this example, it is worth noting that, when analyzing the third call to function `f` at line 6, the reusability condition holds at line 5, so the summary can be reused for this particular call.

We propose a more relaxed condition for reusing function summaries. The new condition needs to verify only (a sound over-approximation of) the read memory locations $R^\sharp$ of the function in order to update the values of (a sound over-approximation of) the written memory locations $W^\sharp$. More precisely, given a statement stmt that does not allocate new memory locations, then its analysis summary is a tuple $(\mathcal{I}^\sharp \downarrow R^\sharp, R^\sharp, \mathcal{O}^\sharp \downarrow W^\sharp, W^\sharp)$: When analyzing stmt with a new input state $\mathcal{I}^\sharp_{\text{new}}$, we only check whether $\mathcal{I}^\sharp_{\text{new}}$ and $\mathcal{I}^\sharp$ are equal for the values restricted to $R^\sharp$. If so, we can reconstruct the output state $\mathcal{O}^\sharp_{\text{new}}$ by merging the values of written memory locations $W^\sharp$ of $\mathcal{O}^\sharp$ with the values of unmodified memory locations $\text{dom}(\mathcal{I}^\sharp_{\text{new}}) \setminus W^\sharp$ of $\mathcal{I}^\sharp_{\text{new}}$. The assumption that stmt does not allocate new memory locations is relaxed later in this section to handle dynamic memory allocation. More formally, if the condition $\mathcal{I}^\sharp_{\text{new}} \downarrow R^\sharp = \mathcal{I}^\sharp \downarrow R^\sharp$ holds before analyzing stmt, then its output state $\mathcal{O}^\sharp_{\text{new}}$ is $\mathcal{O}^\sharp \downarrow W^\sharp \sqcap^\sharp (\mathcal{I}^\sharp_{\text{new}} \downarrow (\text{dom}(\mathcal{I}^\sharp_{\text{new}}) \setminus W^\sharp))$.

▶ **Example 12.** Consider again Example 11. We can infer the following analysis summary of function `f` at line 4, along with its set of read and written memory locations.

$$(\mathcal{I}^\sharp_4 \downarrow R^\sharp_4 = \{p \mapsto \{\&a\}\},\ R^\sharp_4 = \{p\},\ \mathcal{O}^\sharp_4 \downarrow W^\sharp_4 = \{a \mapsto 1\},\ W^\sharp_4 = \{a\}).$$

It is worth noting that $\{a \mapsto 0\}$ is not part of the read input state anymore. When analyzing the call to function `f` at line 5 after having patched the code, we must only check whether $\{a \mapsto 1, p \mapsto \{\&a\}\} \downarrow \{p\}$ is equal to $\{p \mapsto \{\&a\}\}$. By definition of the projection, the condition indeed holds. Therefore, we can build the output state $\{a \mapsto 1\} \sqcap^\sharp (\{a \mapsto 0, p \mapsto \{\&a\}\} \downarrow \{p\})$, which is $\{a \mapsto 1, p \mapsto \{\&a\}\}$.

This relaxed condition allows us to reuse the analysis summary of the first call to function `f` at line 4 when analyzing the second call at line 5. It still allows for reusing it at line 6 as well. Additionally, the memory overhead of the incremental analysis is now reduced since the function summary only stores the projected values of the input and output states.

Extension to Dynamic Memory Allocation

In the case of programs that dynamically allocate memory on the heap, the condition of summary reuse on read memory locations is not sufficient to ensure soundness. Consequently, to preserve soundness, functions that allocate memory dynamically and every caller of these functions will not benefit from summaries.

▶ **Example 13.** Consider for instance the following program:

```
1 void f(int **p) { *p = malloc(sizeof(int)); }
2 void main() {
3     int *a;
4     f(&a);
5     f(&a);
6 }
```

Let μ be the allocation site at the call of malloc_μ in function `f`. We infer the following summary for the analysis of function `f` at line 4:

$$(\mathcal{I}^\sharp_4 \downarrow R^\sharp_4 = \{p \mapsto \{\&a\}\},\ R^\sharp_4 = \{p\},\ \mathcal{O}^\sharp_4 \downarrow W^\sharp_4 = \{a \mapsto \langle \mu, 0 \rangle\},\ W^\sharp_4 = \{a\})$$

When analyzing the call to function `f` at line 5, we check the condition on read memory locations and find that it holds. Therefore, the reconstructed output state is $\{a \mapsto \langle \mu, 0\rangle, \langle \mu, 0\rangle \mapsto Z_0^\sharp\}$. This output state is only correct if we assume that the number of allocations allowed on the site μ is 1, i.e., assuming the call to `f` at line 5 would not generate a new memory location. However, if we assume that $k \geq 2$, the reconstructed output for the call to function `f` at line 5 is incorrect, since the analysis of this call should have generated a new memory location $\langle \mu, 1\rangle$ and updated the value of a to the correct address $\langle \mu, 1\rangle$.

The issue with dynamic memory allocation comes from the condition on read memory locations: It does not capture the allocation of new memory locations. To address this issue, we need to verify that, given an abstract input state $\mathcal{I}^\sharp_{\text{new}}$, we would allocate or reuse the same abstract memory locations (def. 7) as with the saved input state $\mathcal{I}^\sharp$. More formally, we want to verify that the value of the next$^\sharp$ function is the same when applied to $\mathcal{I}^\sharp_{\text{new}}$ and $\mathcal{I}^\sharp$ for each allocation site μ of a given statement. Such a verification is only necessary when a statement stmt dynamically allocates memory, i.e., when $A^\sharp \neq \emptyset$.

▶ **Example 14.** Consider Example 13 again. We can infer the following summary for the analysis of function `f` at line 4, along with the set of read, written and allocated memory locations.

$$(\mathcal{I}_4^\sharp \downarrow R_4^\sharp = \{p \mapsto \{\&a\}\},\ R_4^\sharp = \{p\},\ \mathcal{O}_4^\sharp \downarrow W_4^\sharp = \{a \mapsto \langle \mu, 0\rangle\},\ W_4^\sharp = \{a\},$$
$$\mathcal{O}_4^\sharp \downarrow A_4^\sharp = \{\langle \mu, 0\rangle \mapsto Z_0^\sharp\},\ A_4^\sharp = \{\langle \mu, 0\rangle\})$$

When analyzing the call to function `f` at line 5, we also need to verify whether the input state allocates the same memory locations as the saved input state, i.e., $\forall \langle \mu, \_\rangle \in A_4^\sharp,\ \text{next}^\sharp(\mu, \mathcal{I}_4^\sharp) = \text{next}^\sharp(\mu, \mathcal{I}_5^\sharp)$. The above condition holds only if the number of allocations allowed on the site μ is 1 or no allocation has been made on the site μ with the saved input state $\mathcal{I}_4^\sharp$.

The following theorem expresses the soundness property related to function summaries.

▶ **Theorem 15** (Safe approximation of the computation of a statement)**.** *Given a statement* ***stmt****, a previously computed abstract state* $\mathbb{S}^\sharp[\![\textbf{\textit{stmt}}]\!]\mathcal{I}^\sharp_{old} = \mathcal{O}^\sharp_{old}$*, reusing the analysis summary of* ***stmt*** *with a new abstract input state* $\mathcal{I}^\sharp_{new}$ *is a safe approximation of the computation of* ***stmt*** *if* $\mathcal{I}^\sharp_{new}$ *and* $\mathcal{I}^\sharp_{old}$ *agree on the values of the read memory locations and the set of memory locations allocated by* ***stmt*** *with* $\mathcal{I}^\sharp_{old}$ *is the same as the set of memory locations allocated by* ***stmt*** *with* $\mathcal{I}^\sharp_{new}$*. More formally, let* $\mathcal{I}^\sharp_{old}, \mathcal{I}^\sharp_{new} \in \mathbb{M}^\sharp$ *be two abstract input states, and let* $R^\sharp := Read^\sharp[\![\textbf{\textit{stmt}}]\!]\mathcal{I}^\sharp_{old}, W^\sharp := Write^\sharp[\![\textbf{\textit{stmt}}]\!]\mathcal{I}^\sharp_{old}$, $A^\sharp := Alloc^\sharp[\![\textbf{\textit{stmt}}]\!]\mathcal{I}^\sharp_{old}$, *and* $\mathcal{O}^\sharp_{old} := \mathbb{S}^\sharp[\![\textbf{\textit{stmt}}]\!]\mathcal{I}^\sharp_{old}$*:*

$$\begin{aligned} &\textit{if } \mathcal{I}^\sharp_{new} \downarrow R^\sharp = \mathcal{I}^\sharp_{old} \downarrow R^\sharp \textit{ and } \forall \langle \mu, \_\rangle \in A^\sharp, next^\sharp(\mu, \mathcal{I}^\sharp_{new}) = next^\sharp(\mu, \mathcal{I}^\sharp_{old}) \\ &\textit{then } \mathbb{S}[\![\textbf{\textit{stmt}}]\!]\ \gamma(\mathcal{I}^\sharp_{new}) \\ &\quad \subseteq \gamma((\mathcal{O}^\sharp_{old} \downarrow (W^\sharp \cup A^\sharp)) \sqcap^\sharp (\mathcal{I}^\sharp_{new} \downarrow (dom(\mathcal{I}^\sharp_{new}) \setminus (W^\sharp \cup A^\sharp)))). \end{aligned}$$

Proof. The interesting part of the proof consists of showing that only the values of written and allocated memory locations are modified by the computation of a statement. This means that the values of memory locations that are neither written nor allocated are left unmodified. Additionally, the values of written and allocated memory locations must over-approximate the values of the corresponding memory locations in the concrete computation. The complete proof is available in the extended version. ◀

6 Incremental Analysis with Loop Invariant Reuse

The previous section has shown that, whenever one of the conditions about statements and input states is not satisfied, we need to recompute the function summaries for a given function. This section now shows how we can reuse loop invariants to accelerate the analysis of loops, even if no function summary is available or is reusable. Such an approach was already used in [23] to accelerate the analysis of loops when analyzing nested loops, but without formal details, nor proofs, nor detailed experimental evaluation. This approach works for incremental and non-incremental analyses. We first introduce the program matching process that identifies the loop invariant for reuse. We then show how such an invariant accelerates the analysis of loops and how we can limit the loss of precision induced by such reuse. Finally, we show the soundness and termination of the approach.

Given two program versions P_{new} and P_{old}, we aim to identify similar loop statements in both versions. To this end, we syntactically sort its loops in every function, and we match the loop of index n in function `f` in P_{new} with the loop of same index n in the function with the same name `f` in P_{old}. This matching might be incorrect, typically when one loop statement has been removed and another one has been added in the same function. However, it suffices to reuse some loop invariants from the previous analysis. Furthermore, an incorrect matching does not impact soundness, as demonstrated later in this section.

▶ **Example 16.** Consider the following program and its associated patch:

```
1  void loop() {
2    int i = 0;
3  - while (i < 10)
4  + while (i < 11)
5      i++;
6  }
7  void main() { loop(); }
```

In this example, the loop statement in function `loop` has been modified by changing its upper bound from 10 to 11. Although the loop's condition on both versions of the program are syntaxically different, we consider them as similar since they appear in the same function in the same order. We now know that such a modification invalidates the summaries computed for functions `loop` and `main`. However, we can still reuse the invariant computed for the loop statement in the original analysis of function `loop` in order to accelerate the analysis of the patched version. We now show how we can reuse its loop invariant. Assume the use of the interval domain, as defined in Fig. 1 for the abstract domain of the integer values, and the use of the standard widening operator over intervals at the loop head in order to accelerate convergence when computing the fixpoint. The analysis of the program before the modification of the loop statement is as follows:

$$\text{let } M^\sharp = \{i \mapsto [0,0]\} \text{ in } \mathbb{S}^\sharp[\![\textbf{while } (i<10) \textbf{ do } i := i+1;]\!]M^\sharp \triangleq \bowtie^\sharp_{i<10}(\text{lim } F^\sharp)$$
$$\text{where } F^\sharp(\mathcal{I}^\sharp) \triangleq \mathcal{I}^\sharp \nabla (M^\sharp \sqcup^\sharp \mathbb{S}^\sharp[\![i := i+1;]\!] \circ \bowtie^\sharp_{i<10} \mathcal{I}^\sharp)$$

We start the iteration with the initial abstract state $\bot^\sharp$, and then iterate the abstract semantics of the loop until reaching a post fixpoint. Here are the obtained abstract states:

$$F^{\sharp 0} \triangleq \bot^\sharp \nabla (\{[0,0]\} \sqcup^\sharp \bot^\sharp) = \{[0,0]\}$$
$$F^{\sharp 1} \triangleq \{[0,0]\} \nabla (\{[0,0]\} \sqcup^\sharp \{[1,1]\}) = \{[0,+\infty]\}$$
$$F^{\sharp 2} \triangleq \{[0,+\infty]\} \nabla (\{[0,0]\} \sqcup^\sharp \{[1,10]\}) = \{[0,+\infty]\}$$

The iteration sequence converges to the invariant $[0, +\infty]$ in three iterations. Now, we apply the patch to the program and analyze the modified version. Our strategy for loop invariant reuse consists in starting the iteration sequence with the invariant previously computed (which we will refer to as $\mathcal{I}^\sharp_{\text{prev}}$) instead of starting it from the initial abstract state $\bot^\sharp$. With such a strategy, the iteration sequence of the analysis of the loop for the modified program is as follows: $F^{\sharp 0} \triangleq (\mathcal{I}^\sharp_{\text{prev}} = \{[0, +\infty]\}) \nabla (\{[0, 0]\} \sqcup^\sharp \{[1, 11]\}) = \{[0, +\infty]\}$. The iteration sequence converges to the invariant $[0, +\infty]$ after a single iteration: We successfully reduced the number of iterations needed to converge to the invariant by reusing $\mathcal{I}^\sharp_{\text{prev}}$. Such a reduction might seem not important at a first glance since it "only" removes two iterations, but their accumulation becomes significant when analyzing large programs as shown in Section 7, since such fixpoint computations are frequent and time-consuming.

We now show the impact of reusing loop invariants on the precision of the analysis. We know that the invariant computed for a loop statement in the previous analysis is a sound over-approximation of the concrete least fixpoint. Consequently, it contains values that are not necessarily used by the loop statement. However, such values might have changed in the new version of the program, and reusing the invariant might propagate imprecise values. To limit such an imprecision, we only reuse the values of memory locations that are used inside the loop. Such memory locations are defined as follows:

▶ **Definition 17** (Set of abstract used memory locations).

$$Used^\sharp[\![\textbf{\textit{stmt}}]\!]M^\sharp \triangleq Read^\sharp[\![\textbf{\textit{stmt}}]\!]M^\sharp \cup Write^\sharp[\![\textbf{\textit{stmt}}]\!]M^\sharp \cup Alloc^\sharp[\![\textbf{\textit{stmt}}]\!]M^\sharp.$$

Given a former invariant $\mathcal{I}^\sharp_{\text{prev}}$, an abstract input state $M^\sharp$ and a loop statement **stmt**, we reduce the invariant $\mathcal{I}^\sharp_{\text{prev}}$ to $\mathcal{I}^\sharp_{\text{red}}$ in the following way:

$$\text{let } U^\sharp = Used^\sharp[\![\textbf{stmt}]\!]M^\sharp \text{ in } \mathcal{I}^\sharp_{\text{red}} \triangleq (\mathcal{I}^\sharp_{\text{prev}} \downarrow U^\sharp) \sqcap^\sharp (M^\sharp \downarrow (\text{dom}(M^\sharp) \setminus U^\sharp)).$$

Reducing $\mathcal{I}^\sharp_{\text{prev}}$ to $\mathcal{I}^\sharp_{\text{red}}$ lets us also reduce the memory space used when storing invariants in the cache. This reduction is important when analyzing large programs.

▶ **Example 18.** Suppose we are analyzing the loop statement **while** $(i < 10)$ **do** $i := i + 1$ with the invariant $\mathcal{I}^\sharp_{\text{prev}} = \{i \mapsto [0, +\infty], j \mapsto [1, 1]\}$. Suppose the initial abstract state before the loop is $M^\sharp = \{i \mapsto [0, 0], j \mapsto [0, 0]\}$, where the value of j differs from $\mathcal{I}^\sharp_{\text{prev}}$. We would like to reuse $\mathcal{I}^\sharp_{\text{prev}}$ to accelerate the analysis of the loop, but we would also like to avoid propagating the value of j, as its value would make the analysis less precise.

Initially, the iteration sequence of the analysis of the loop converges to the invariant $\{i \mapsto [0, +\infty], j \mapsto [-\infty, 1]\}$, since the widening operator would replace the lower bound of j with $-\infty$ because j is unstable and decreasing. We show the part of the iteration sequence that propagates the value of j:

$$F^{\sharp 1}(\{j \mapsto [1, 1]\}) \triangleq \{j \mapsto [1, 1]\} \nabla (\{j \mapsto [0, 0]\} \sqcup^\sharp \{j \mapsto [1, 1]\}) = \{j \mapsto [-\infty, 1]\}.$$

However, by forgetting the value of j from $\mathcal{I}^\sharp_{\text{prev}}$, the iteration sequence of the analysis of the loop converges to the invariant $\{i \mapsto [0, +\infty], j \mapsto [0, 0]\}$. The idea is to reuse only the values of memory locations used by the statement. The reduced invariant is as follows:

$$\begin{aligned}\mathcal{I}^\sharp_{\text{red}} =& (\{i \mapsto [0, +\infty], j \mapsto [1, 1]\} \downarrow \{i\}) \sqcap^\sharp (\{i \mapsto [0, 0], j \mapsto [0, 0]\} \downarrow \{j\}) \\ =& \{i \mapsto [0, +\infty], j \mapsto [0, 0]\}\end{aligned}$$

Nevertheless, some imprecision cannot be avoided when reusing loop invariants. Consider the following example:

▶ **Example 19.** Suppose we analyzed the loop statement **while** $(i < 10)$ **do** $i := i + 1$ where the initial abstract state is $M_1^\sharp = \{i \mapsto [0, 0]\}$ and we obtained the invariant $\mathcal{I}^\sharp_{\text{prev}} = \{i \mapsto [0, +\infty]\}$. We now analyze the same loop with $\mathcal{I}^\sharp_{\text{prev}}$ but with a different initial abstract state $M_2^\sharp = \{i \mapsto [1, 1]\}$, where the value of i differs from the two abstract states. The analysis of the loop converges to the invariant $\{i \mapsto [0, +\infty]\}$ when reusing $\mathcal{I}^\sharp_{\text{prev}}$, which is less precise than the invariant $\{i \mapsto [1, +\infty]\}$ that we would have obtained if we had started the iteration sequence from $\bot^\sharp$. Such an imprecision is inevitable when starting the iteration sequence from a correct but less precise invariant, i.e., $\{i \mapsto [1, +\infty]\} \sqsubseteq^\sharp \mathcal{I}^\sharp_{\text{prev}}$.

Finally, we show the soundness and termination of the approach.

▶ **Theorem 20** (Soundness and Termination)**.** *For any loop statement* ***while*** $(e \bowtie 0)$ ***do*** s*, abstract state* $M^\sharp$ *and invariant* $\mathcal{I}^\sharp$*, the limit of the iteration sequence of the analysis of* ***while*** $(e \bowtie 0)$ ***do*** s *starting from* $\mathcal{I}^\sharp$ *and such that* $\bowtie^\sharp_e(lim\ F^\sharp)$ *where* $F^{\sharp 0} = \mathcal{I}^\sharp \nabla (M^\sharp \sqcup^\sharp \mathbb{S}^\sharp[\![s]\!] \circ \bowtie^\sharp_e \mathcal{I}^\sharp)$ *is sound and always terminates.*

Proof. The iteration sequence starting from an abstract state $\bot^\sharp$ is known to converge to a sound approximation of the concrete least fixpoint. However, we can start the iteration sequence from an arbitrary abstract state $\mathcal{I}^\sharp$: As long as the stability test $F^\sharp(\mathcal{I}^\sharp) \sqsubseteq^\sharp \mathcal{I}^\sharp$ which verifies if the invariant is a post-fixpoint is correct, the invariant $\mathcal{I}^\sharp$ is a sound approximation of the concrete least fixpoint, regardless of the steps of the iteration sequence. ◀

In summary, this approach is complementary to function summaries, as it does not require any condition to be satisfied to reuse the previous invariant. We have shown that reusing the previous invariant makes the analysis faster by reducing the number of iterations before reaching a stable iterate but may lose precision for some cases. Section 7 will show that, in practice, reusing loop invariants significantly accelerates the analysis of loops and does not impact the precision of the analysis.

7 Implementation and Evaluation

This section presents our implementation of the proposed techniques for function summaries and loop invariants, built on top of the Eva [4, 6] plug-in of Frama-C [3, 19]. It is evaluated on a benchmark of open-source C programs. The incremental version of Frama-C and the set of programs used for the evaluation are available as supplementary material.

7.1 Implementation

The Eva plug-in already computes function summaries without support for dynamic memory allocation through the Memexec [37] cache. Moreover, Frama-C already provides a mechanism to store analysis results in a dedicated file [5]. Last, Frama-C also provides an abstract syntax tree (AST) diff tool to compare the ASTs of two relatively similar programs. In our work, we first extended the Memexec cache to support dynamic memory allocation and to compute and store loop invariants. Then, the saved function summaries and loop invariants are reloaded in subsequent analysis runs after comparing the abstract syntax trees of the analyzed programs. Finally, function summaries impacted by the changes are invalidated before running the analysis.

7.1.1 Function Summaries

Eva's Memexec computes and stores function summaries after analyzing a function with a new analysis context. Additionally, Memexec does not bound the number of summaries stored in the cache. We extended Memexec to support dynamic memory allocation by storing the abstract set of memory locations allocated by the function in the summary. However, the reusability condition differs slightly from the one presented in Section 5. In fact, Eva features a mechanism to deallocate memory locations. To do so, Eva removes the memory location from the abstract state when it is freed. More precisely, suppose that we want to analyze a statement $*\&x = \text{malloc}_\mu$ with an abstract input state $M^\sharp$, and we are given a set $A^\sharp$ of memory locations already allocated on the site μ. If Eva finds a memory location $\langle \mu, n \rangle$ in $A^\sharp$ that has been freed, i.e., $\langle \mu, n \rangle \notin \text{dom}(M^\sharp)$, then Eva reuses the memory location $\langle \mu, n \rangle$ instead of generating a new one. This mechanism is particularly useful for avoiding generating new memory locations when the same memory location is allocated and deallocated repeatedly, typically in a loop. It enhances the precision of the analysis by delaying as much as possible the generation of weak memory locations. We leverage this mechanism to reuse function summaries by checking whether the set of allocated memory locations $A^\sharp$ generated by the analysis of the function is reusable or not given a new abstract input state $\mathcal{I}^\sharp_{\text{new}}$, i.e., $A^\sharp \sqcap \text{dom}(\mathcal{I}^\sharp_{\text{new}}) = \emptyset$. This condition is sound with respect to the current handling of dynamic memory allocation in Eva. However, the formal proof of the soundness of this condition is left for future work.

7.1.2 Loop invariants

Contrary to function summaries, Eva had no mechanism to reuse loop invariants. Therefore, we extended it to do so: First, we modified AST Diff to build the correspondence between the loops of the functions of both programs. Then, we added a cache of loop invariants for each loop statements of each function in Memexec. The reusability condition of loop invariants presented in Section 6 states that any loop invariant can be soundly reused. We are left with the task of finding the loop invariant to reuse when multiple invariants are stored in the cache. Finding such an invariant is challenging because, although every choice maintains soundness, it has an impact on the performance and the precision of the analysis. More precisely, we want to find the closest invariant to the one we are looking for. To do so, we decided to store separately new invariants when the analysis of the current function is done with a new analysis context. The analysis context is defined here as the callstack and the abstract values of the argument of the function. When, the same analysis context is encountered, the new invariant is merged using the abstract join $\sqcup^\sharp$ operator with the one stored in the cache.

7.2 Evaluation

Research questions

We evaluate our implementation to address the following research questions:

RQ1: Is incremental analysis more time-efficient than a full analysis?

RQ2: What is the impact of incremental analysis on max memory usage and on-disk storage?

RQ3: What is the impact of incremental analysis on the precision of the analysis?

RQ4: What is the effect of cache loading and AST comparison on analysis time?

Table 1 Program description.

Program	# Commit	LoC	Function	Loop	Diff Summary
PolarSSL	41	28K	818-822	293-302	52 files, 887 (+), 259 (−)
Monocypher	107	7K	660-823	210-244	147 files, 15348 (+), 3395 (−)
Chrony	119	25K	206-265	125-162	109 files, 10754 (+), 1698 (−)

Experimental Setup

Our experiments focus on analyzing three real-world programs: PolarSSL [1], Monocypher [2] and Chrony [3], issued from the Open Source Case Studies (OSCS) [4]. Table 1 shows the total number of commits analyzed, the average number of lines of code (LoC), the minimum and maximum number of functions and loops in the program, and the summary of the diff between the first and last commit. The diff summary shows the total number of files modified, the total number of lines added and removed. All programs, written in C, were already pre-configured and modified to be analyzable with Eva for a classical non-incremental analysis. For our evaluation, we simulate a development scenario where developers modify the source code of these programs and push the changes to the repository. The commits consist of real modifications made by the developers of both projects. Each code commit triggers an analysis run in the continuous integration (CI) pipeline. The analysis process reuses function summaries and loop invariants computed during the previous analysis run. Every commit includes modifications affecting the source code. The code source in OSCS contains annotations made by the maintainers of the repository. These annotations are mandatory to guide the analysis of Eva and ensure it completes in a reasonable time. We extracted these annotations as a patch and applied them to each commit before running the analysis. Since a commit involves changes to the source code, we need to ensure for each commit that the patch applies correctly, that Frama-C can parse the modified code, and that Eva's analysis completes successfully. We took the maximum set of commits that successfully passes these checks for each program. Adding the commits where any of these checks failed would involve changes to the annotations or even the analysis parameters for each additional commit. No additional code annotations were needed for the incremental analysis. Our experiments are executed on a machine with an Intel Core i7-12800H CPU clocked at 2.4 GHz and 64 GB of RAM. The machine runs Manjaro Linux 6.6. We used Frama-C version 28.1 extended with our development for incremental analysis. The analysis parameters were set to the values already used in OSCS for non-incremental analysis. We also set the maximum number of allocation per site k_{lim} to 2 with `-eva-mlevel 2` option and we disabled the possibility of `malloc` to fail with `-eva-no-alloc-returns-null` option.

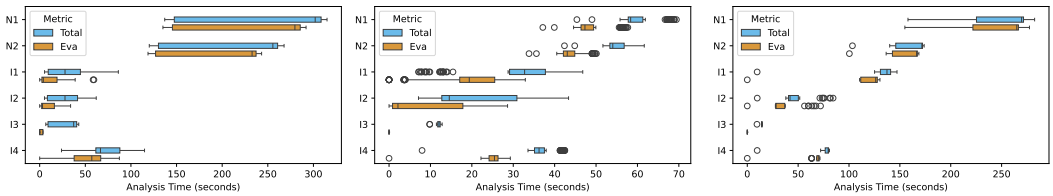
7.3 Results

To answer the research questions RQ1–RQ4, we evaluate the performance and precision of our approach by focusing on criteria such as analysis time, memory consumption, and number of alarms generated on six different analysis settings for each version of PolarSSL,

[1] `https://github.com/Mbed-TLS/mbedtls`
[2] `https://github.com/LoupVaillant/Monocypher`
[3] `https://github.com/mlichvar/chrony`
[4] `https://git.frama-c.com/pub/open-source-case-studies`

Table 2 Analysis settings.

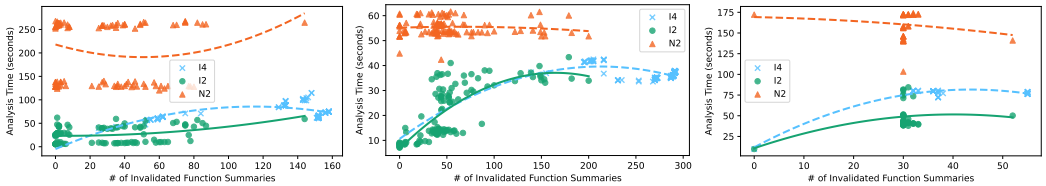
Setting	Type	Version Reused	Approach
N1	Full	None	Summaries
N2	Full	None	Summaries, Invariants
I1	Incremental	Immediate previous	Summaries
I2	Incremental	Immediate previous	Summaries, Invariants
I3	Incremental	Same version	Summaries, Invariants
I4	Incremental	First version	Summaries, Invariants



(a) Monocypher. **(b)** Chrony. **(c)** PolarSSL.

Figure 8 Analysis time measurement results.

Monocypher and Chrony as shown in Table 2. Full Analyses (N1 and N2) are used as a reference for the comparison with Incremental Analyses (I1, I2, I3, and I4). Even though N1 and N2 do not reuse any previous results, they still benefit from summaries and invariants computed during the ongoing analysis. Incremental Analyses (I1 and I2) reuse the results of the immediate previous version of the program when they are available. For the first version of the program, we run a full analysis similar to N1 (resp. N2) for I1 (resp. I2). The first analysis results are then reused by the second version, and so on. The difference between N1 and N2 (resp. I1 and I2) is the reuse of loop invariants. It allows us to evaluate the impact of reusing loop invariants on the analysis results. Incremental Analysis (I3) sets the limit of the maximum efficiency that can be achieved by our approach. We run first a full analysis of each version of the programs and then we incrementally analyze each version with itself. Therefore, no changes are made to the programs and all summaries and invariants are reused. Incremental Analysis (I4) sets the limit of the minimum efficiency that can be achieved by our approach. We run first a full analysis on the first version (v_0) of the programs and then we incrementally analyze each version ($v_1, ..., v_n$) with the results from the first version (v_0). The analysis of v_0 reusing the results of v_0 represents the best case scenario, whereas the analysis of v_n reusing the results of v_0 where n is the last version of the program represents the worst case scenario. Figure 8 shows the analysis time measurement for each program and analysis setting. We separate the total analysis time from Eva's analysis time. The latter is the abstract interpretation per se while the former includes the time spent on parsing the program, loading the cache and computing the impact of changes. Eva's analysis correctly represents the impact of the approach on the analysis time. The difference between the two metrics represents the overhead of incremental analysis and can be further reduced by optimizing cache loading and program matching. Circles represent outliers, i.e., data points that are significantly different from the rest of the data. Outliers such as sub-second Eva analysis time are not removed from the box plot as they are not considered as anomalies. We only remove the analysis of the first versions of each program in I1 and I2 because they cannot reuse any previous results and therefore they are not relevant. Figure 9 shows the



(a) Monocypher. **(b)** Chrony. **(c)** PolarSSL.

Figure 9 Impact of invalidated function summaries on analysis time.

Table 3 Summary of analysis results across programs and analysis settings.

	Analysis Setting	Total Alarms	Total Iter.	Max Mem. (MB)	Cache Load(s)	AST Diff.(s)	Cache Size(MB)
P	N1	29097	839265	1161			89.40
	I1	∼	**1.17x**	0.42x	7	4	0.63x
	N2	∼	1.62x	1.01x			1.01x
	I2	0.99x	4.00x	0.50x	5	4	0.68x
M	N1	25356	14478001	2981			130.42
	I1	0.99x	24.46x	0.64x	7	2	0.65x
	N2	**1.01x**	1.34x	0.94x			0.98x
	I2	**1.01x**	35.22x	0.64x	7	2	0.63x
C	N1	404748	1402787	1249			100.75
	I1	0.99x	1.81x	0.76x	2	4	0.85x
	N2	0.99x	1.6x	0.97x			0.99x
	I2	0.99x	6.9x	0.77x	2	4	0.90x

relationship between the number of invalidated function summaries and the analysis time. It also indicates the impact of the changes made to the program by the amount of invalidated summaries. Table 3 presents a comprehensive comparison between a full analysis and an incremental analysis across several performance metrics. We use N1 as the reference for the comparison with the other analysis settings (I1, N2 and I2). We display the ratio (Full Analysis / Incremental Analysis) for most cases except cache loading and AST Diff time. Cases with a significant difference are highlighted in bold. Cases annotated with ∼ indicate that the value is the same as the reference. Cases without values indicate a non-applicable metric. The first column indicates the program analyzed with **P** for PolarSSL, **C** for Chrony, and **M** for Monocypher. The second column indicates the analysis setting. The third column shows the total number of alarms generated by the analysis of all versions. The fourth column shows the total number of iterations required before convergence of all versions. The fifth column shows the peak memory usage. The sixth and seventh columns show the average time spent on loading the cache and computing the impact of changes, respectively. The last column shows the maximum size of the cache file.

RQ1: Time Efficiency

Figure 8 shows that Incremental Analysis I1 (resp. I2) is faster than Full Analysis N1 (resp. N2) for all programs. The efficiency of reusing function summaries alone is shown by N1 and I1. The further reduction in analysis time when reusing loop invariants is shown by

the differences N2-N1 and I2-I1. Moreover, Table 3 shows a significant reduction of the total iteration required to converge to a fixpoint when reusing loop invariants. Additionally, we observe some cases with Chrony where I1 is slower than N1. This is due to the invalidation of a large number of summaries. Consequently, the analysis does not benefit from previous results and the total analysis time increases due to the overhead of loading the cache and computing the impact of changes. Moreover, we observe that Incremental Analysis I1 is slower than Full Analysis N2 for PolarSSL. The number of iterations required to converge shown in Table 3 is higher for I1 than N2. This indicates that reusing function summaries is not always sufficient and the combination of summaries and loop invariants is necessary to achieve the best performance. We observe in Figure 8 for I3 that Eva's analysis time is instantaneous as no summaries are invalidated. Consequently, the total analysis time is only spent on loading the cache and computing the impact of changes. We observe in Figure 9 for I4 that the analysis time increases as the number of invalidated summaries increases. However, I4 is still more time-efficient than N2 in most cases. Nevertheless, the number of invalidated summaries is still not the best indicator – although better than the raw commit diff – to predict the outcome the analysis time. Indeed, the analysis time of each function can be different depending on its complexity. For example, the analysis of a function with a large number of loops or having deeply nested function calls is more time-consuming than the analysis of a logging function. The significant reduction in analysis time shows the efficiency of our incremental approach, making it highly suitable for use in CI/CD environments where quick feedback on code changes is essential.

RQ2: Max Memory Usage and On-Disk Storage

Table 3 shows that Incremental Analysis uses more memory than Full Analysis for all programs. This increase in memory usage is expected as we trade memory for time efficiency. Additionally, adding summaries and invariants to the cache increases the size of the dedicated file on disk. However, we believe that the increase in memory consumption and on-disk storage of Incremental Analysis is still acceptable and worth the time efficiency gained as they remain stable during the analysis. Moreover, the number of summaries and invariants we store is currently unbounded, thus, it is possible to modify this strategy and dynamically adjust this number in case of memory issues.

RQ3: Precision

We compare the total number of alarms generated by each analysis setting to N1 to assess the precision of our approach. We observe three different cases in Table 3: First, all incremental analysis settings I2 for all programs generated more alarms than N1. This behavior is expected as the reuse of loop invariants might introduce imprecision in the analysis. Although, we observe only a very slight increase in the number of alarms. Second, the number of alarms generated by I1 for Chrony and Monocypher also increases compared to N1. This increase is directly related to the reuse of function summaries involving dynamic memory allocation (summaries of function without dynamic memory allocation are not impacted). More precisely, we found that reusing summaries that contain weak memory locations introduces imprecision in the analysis in some cases because a full Eva analysis could have reused a strong memory location instead. This behavior is related to the definition of the next function which does not take into account memory deallocation, therefore, when the maximum number of allocation per site k_{lim} is reached, the function next always points to the last weak memory location. Third, the number of alarms generated by N2 and I2 for Monocypher decreases compared

to N1. Although unexpected, this behavior is related to the non-monotonicity property of widenings. More precisely, the reuse of loop invariants during the analysis of loops leads to the generation of more precise values. The non-monotonicity property of widening is studied in [9], and we have therefore an experimental evidence of its impact on real-world programs instead of crafted examples. Overall, this indicates that the incremental approach maintains a high level of precision, ensuring that the incremental approach is about as good as the non-incremental approach.

RQ4: Cache Loading and Program Matching

Table 3 shows that the overhead of loading the cache and computing the impact of changes is acceptable. The actual cases where such overhead is significant are when Eva's analysis time is very fast. Nonetheless, there is still room for improvement such as a partial or deferred cache loading and the use of more efficient program matching algorithms.

Summary

Overall, our approach offers substantial benefits in terms of time efficiency, with an acceptable memory usage and on-disk storage cost, and a slight overhead in loading the cache and computing the impact of changes. The analysis remains precise enough and even more precise in some cases. The reuse of loop invariants further enhances performance. These results suggest that incremental analysis is a valuable technique for improving the efficiency of static analysis in CI/CD environments.

Threats to Validity

We identify the following threats to the validity of our results: We reuse mostly the same analysis parameters as the ones already configured for a non-incremental analysis. Therefore, the effects of changing these parameters have not been explored: Our results could vary significantly when modifying them. Even though we conduct our experiments on three real-world programs, the generalizability of our results to other programs remains uncertain. The specific characteristics of these programs could influence the results, making their applicability to other programs unknown.

8 Related work

Incremental static analysis is a technique that has gained significant attention in recent years due to the need for fast and efficient analyses of large growing codebases. This technique speeds up non-incremental analyses by reusing previous analysis results and avoiding redundant work. This section reviews the current state of the art in incremental static analysis by abstract interpretation and highlights the gap that our work fills.

Incremental Modular Analysis. Modular analyses [12], based on summaries [7, 36, 35, 22, 27, 8] enable incremental analysis by analyzing only the modules that are affected by the changes in the program. Work in [35] proposes an approach to make modular static analysis incremental by exploiting dependencies between modules. They show that the dependencies computed during the analysis can be used to bound the impact of a change. Compared to this work our approach is more generic and does not require the original analysis to be modular. Work in [16] proposes a modular incremental analyzer capable of performing fine-grain incremental analysis across modular program partitions. However, their approach

assumes that the program is represented as Constrained Horn clauses and the analysis is done on the Horn clause representation. Compared to this work, our approach is more generic and does not require an intermediate representation far from the original program. Commercial tools like Coverity [22] and Infer [27] support modular incremental analysis on large codebases. However, they do not provide a soundness guarantee, which makes them unsuitable for critical software. In particular, Infer only shows the new alarms that are generated by the changes in the code after the analysis. More precisely, Infer only analyses the functions that are impacted by the changes in the code, and it does not guarantee that the analysis is sound. A shared problem with modularity through summaries is the loss of precision induced by the formalism used. A safe modular analysis implies a summary capable of representing all possible call contexts. However, computing a precise summary is costly, which leads to a trade-off between precision and cost. Our approach is inspired by summaries and guarantees safety while maintaining a high level of precision in the analysis.

Incremental Analysis Framework. Recent work has incrementalized several classes of program analyses. For instance, [1] focuses on dataflow analyses but is limited to distributive lattices, which rules out interval abstract domains. Frameworks such as IncA [33, 31, 32] provide a DSL for defining incremental analyses and algorithms to maintain analysis results. IncA translates the user-defined analysis to an intermediate representation such as graph-patterns and Datalog. It supports incremental control flow and points-to analysis for mbedrr C [33], and strong-update points-to, string, and constant propagation analyses for Java [31, 32]. Code modifications rely on a given AST changes notification, e.g., from IDEs. The incremental strategy maintains results across changes and provides real-time feedback. However, correctness and efficiency depend on the limitations and assumptions of the underlying Datalog solver, which analysis developers must satisfy. For instance, [33] do not support recursive user-defined aggregation, which limits analysis results to powerset lattices only. Later work [31] removes such limitations by providing a new algorithm and the use of widenings for infinite-height lattice, but imposes monotonicity over the aggregation operator and are only efficient for intra-procedural analyses. Work in [32] loosen the monotonicity requirement, although still imposes it in some cases and is efficient for inter-procedural analyses. In contrast, our approach does not target user-defined program analyses, but rather incrementalizes the abstract interpretation framework itself. We do not depend on the underlying abstract domains as long as they are non-relational. Abstract domains only need to be a poset, not a CPO nor a lattice, and abstract operators need not be monotonic – which they are often not, due to widenings and reductions. Moreover, our approach does not incrementally maintain analysis results given a set of changes, but rather runs the analysis of a program with a completely new AST and only reuses previous results when possible. This approach is more suitable for existing whole program analysis frameworks, such as Frama-C [3], Mopsa [24] and Astrée [23], where incrementality has not been considered. Other recent work focuses on incrementalizing existing frameworks. Works in [28, 29] incrementalize a generic local fixpoint solver: They show that the data structures used in the solver can be exploited to limit the part of the program that needs to be reanalyzed. They also support on-demand analyses [14] for which, given a specific query to the analyzer, it only computes the necessary dependencies to answer the query. Work in [30] proposes a new operational semantics that reifies abstract interpretation on a directed acyclic graph. It shows that the dependencies between the nodes in the graph can be exploited to invalidate only the nodes that are affected by the change in the program. Compared to these works, we provide a generic framework that can be built on top of any existing abstract interpreter instead

of incrementalizing a particular framework. Reusing fixpoints for incremental analysis is also a common technique used in incremental analysis. For instance, [26] matches pre- and post-change programs and reuses fixpoints for matched parts. It ensures soundness by reanalyzing the entire program, making it conservative–imprecision from mismatches may propagate. Our method limits fixpoint reuse to loops in modified functions or when function summaries cannot be reused, minimizing imprecision propagation.

Cache Reuse. Caching is pervasive and mandatory in practice for any large scale analysis. Abstract interpreters such as Mopsa [24, 25] and Astrée [23] use caching for function and fixpoint iteration in order to accelerate interprocedural and multi-threaded analyses. The Eva [4] plugin of Frama-C [3] also uses caching for function summaries, which did not support dynamic memory allocation before our extension. However, the reuse of the cache is limited to the ongoing analysis and are not designed to be soundly reused across different analyses. Compared to these works, we provide the conditions under which the cache can be reused or invalidated across different analysis.

9 Conclusion and Future Work

This paper presented a sound, generic approach to incremental static analysis based on reusing results from previous analysis runs. We introduced two complementary approaches: one for computing and reusing function summaries–precise and sound when functions are not affected by changes–and another for reusing loop invariants, which remain sound even when loops are modified, though potentially less precise. Combined, these two approaches can be used to significantly accelerate the analysis of large software. They are beneficial for both ongoing analysis and incremental analysis. They are suitable for critical software and for CI/CD approaches. We have implemented our approach in the Eva plugin of Frama-C and evaluated it on a set of open source programs. Our evaluation shows that our approach can significantly reduce the analysis time of these programs, while maintaining a high level of precision. Our next steps include extending function summaries to handle relational domains. As seen in the semantics of the language, function summaries only compute the set of memory locations inductively from the syntax. However, such a set cannot include by definition the set of memory locations that are related to each other. We already implemented a solution to this problem within Eva. More precisely, we compute the transitive closure of the relation between the read memory locations given by each relational domain used in the analysis. The next step is to formalize this solution and to prove its correctness. Additionally, we plan to extend our formalization to handle more features of the C language, such as pointer arithmetic, recursive function calls and memory deallocations. Moreover, we plan to use more sophisticated techniques for the program matching techniques, allowing us to reuse more analysis results. Finally, we plan to take into account changes in the parameters of the analysis. That is, static analyzers are often used after the initial development phase, typically by a separate team whose goal is to certify the absence of runtime errors. To do so, the analysis parameters are finely tuned incrementally to improve the precision. Since the cost of this process is an order of magnitude higher than the cost of the initial analysis, we aim to adapt our approach to this methodology of incremental tuning of the analysis parameters. However, not only is it crucial to avoid any loss of precision resulting from incrementality, as it currently the case, but we also aim to improve the precision of the analysis.

References

1 Steven Arzt and Eric Bodden. Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*, pages 288–298, Hyderabad India, May 2014. ACM. doi:10.1145/2568225.2568243.

2 Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In Kwangkeun Yi, editor, *Static Analysis*, pages 221–239, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11823230_15.

3 Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the frama-c software analysis platform. *Commun. ACM*, 64(8):56–68, 2021. doi:10.1145/3470569.

4 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*, volume 10145, pages 112–130. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-52234-0_7.

5 François Bobot, André Maroneze, Virgile Prevosto, and Julien Signoles. *The Art of Developing Frama-C Plug-ins*, pages 341–401. Springer International Publishing, July 2024. doi:10.1007/978-3-031-55608-1_7.

6 David Bühler, André Maroneze, and Valentin Perrelle. *Abstract Interpretation with the Eva Plug-in*, pages 131–186. Springer International Publishing, July 2024. doi:10.1007/978-3-031-55608-1_3.

7 Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015. doi:10.1007/978-3-319-17524-9_1.

8 Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011. doi:10.1145/2049697.2049700.

9 Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2022.

10 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '77*, pages 238–252, Los Angeles, California, 1977. ACM Press. doi:10.1145/512950.512973.

11 Patrick Cousot and Radhia Cousot. Constructive Versions of Tarski's Fixed Point Theorems. *Pacific Journal of Mathematics*, 82(1), 1979.

12 Patrick Cousot and Radhia Cousot. Modular Static Program Analysis. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and R. Nigel Horspool, editors, *Compiler Construction*, volume 2304, pages 159–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. doi:10.1007/3-540-45937-5_13.

13 David Delmas and Antoine Miné. Analysis of Software Patches Using Numerical Abstract Interpretation. In Bor-Yuh Evan Chang, editor, *Static Analysis*, volume 11822, pages 225–246. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-32304-2_12.

14 Julian Erhard, Simmo Saan, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani, and Helmut Seidl. Interactive Abstract Interpretation: Reanalyzing Whole Programs for Cheap, November 2022. doi:10.48550/arXiv.2209.10445.

15 Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2642937.2642982.

16 Isabel Garcia-Contreras, Jose F. Morales, and Manuel V. Hermenegildo. Incremental and Modular Context-sensitive Analysis. *Theory and Practice of Logic Programming*, 21(2):196–243, March 2021. doi:10.1017/S1471068420000496.

17 Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988, pages 512–529, March 2004. doi:10.1007/978-3-540-24730-2_38.

18 Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *ACM Comput. Surv.*, 49(2), June 2016. doi:10.1145/2931098.

19 Nikolaï Kosmatov, Virgile Prevosto, and Julien Signoles. *Guide to Software Verification with Frama-C: Core Components, Usages and Applications*. Springer Cham, July 2024. doi:10.1007/978-3-031-55608-1.

20 William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In Stuart I. Feldman and Richard L. Wexelblat, editors, *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM. doi:10.1145/143095.143137.

21 MathWorks. Polyspace. https://www.mathworks.com/products/polyspace.html. Accessed: 2024-12-20.

22 Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, Esec/Fse 2013, pages 554–564, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491411.2501854.

23 Antoine Miné and David Delmas. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In *2015 International Conference on Embedded Software (EMSOFT)*, pages 65–74, Amsterdam, Netherlands, October 2015. IEEE. doi:10.1109/EMSOFT.2015.7318261.

24 Antoine Miné, Abdelraouf Ouadjaout, and Matthieu Journault. Design of a Modular Platform for Static Analysis. In *The Ninth Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, 2018.

25 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static Type Analysis by Abstract Interpretation of Python Programs. In *Leibniz International Proceedings in Informatics (LIPIcs)*, 2020. doi:10.4230/LIPIcs.ECOOP.2020.17.

26 Lawton Nichols, Mehmet Emre, and Ben Hardekopf. Fixpoint reuse for incremental JavaScript analysis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 2–7, Phoenix AZ USA, June 2019. ACM. doi:10.1145/3315568.3329964.

27 Peter W. O'Hearn. Continuous Reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 13–25, Oxford United Kingdom, July 2018. ACM. doi:10.1145/3209108.3209109.

28 Helmut Seidl, Julian Erhard, and Ralf Vogler. Incremental Abstract Interpretation. In Alessandra Di Pierro, Pasquale Malacaria, and Rajagopal Nagarajan, editors, *From Lambda Calculus to Cybersecurity Through Program Analysis*, volume 12065, pages 132–148. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-41103-9_5.

29 Helmut Seidl and Ralf Vogler. Three improvements to the top-down solver. *Mathematical Structures in Computer Science*, 31(9):1090–1134, October 2021. doi:10.1017/S0960129521000499.

30 Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Demanded abstract interpretation. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, PLDI 2021, pages 282–295, New York, NY, USA, 2021. ACM. doi:10.1145/3453483.3454044.

31 Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in datalog. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276509.

32 Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. Incremental whole-program analysis in datalog with lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 1–15, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454026.

33 Tamás Szabó, Sebastian Erdweg, and Markus Voelter. IncA: A DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 320–331, Singapore Singapore, August 2016. ACM. doi:10.1145/2970276.2970298.

34 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, June 1955. doi:10.2140/pjm.1955.5.285.

35 Jens Van der Plas, Quentin Stievenart, Noah Van Es, and Coen De Roover. Incremental Flow Analysis through Computational Dependency Reification. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 25–36, Adelaide, Australia, September 2020. IEEE. doi:10.1109/SCAM51674.2020.00008.

36 Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static race detection on millions of lines of code. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 205–214, Dubrovnik Croatia, September 2007. ACM. doi:10.1145/1287624.1287654.

37 Boris Yakobowski. Fast whole-program verification using on-the-fly summarization. In *Workshop on Tools for Automatic Program Analysis (TAPAS)*, page 2, 2015.