# ZooBerry: A Software Framework for Global Sparse Analyzers and Their Verified Validators

Sungkeun Cho, Jeehoon Kang, Chung-Kil Hur, and Kwangkeun Yi

Seoul National University, Korea

**Abstract.** We report a software framework, called ZooBerry, that fills the gap between static analysis designs (abstract semantics and their soundness proofs) and their faithful yet scalable implementations (industrial-strength global analyzers whose results can be automatically checked correct). The input to ZooBerry is an abstract semantics (an analysis specification) and its soundness proof, both in Coq. For scalable implementation, ZooBerry automatically integrates into the abstract semantics the general sparse analysis technology [27,26] that streamlines both the spatial and temporal footprints of the generated analyzers. For faithful implementation, ZooBerry also generates a verified validator (and its safety proof in Coq) that will check the correctness of each analysis result of the generated sparse analyzer. This validator will continue to check the correctness of posterior changes of the generated analyzers for further cost reduction.

ZooBerry's performance is tested in realistic settings. We have implemented two analyzers for C programs, together with their validators, that detect buffer overrun and format string bugs. Both analyzers' cost performance is like they take 10-20 minutes to globally analyze 100KLoC of C benchmarks. From the format string bug analyzer we were able to find two security vulnerabilities that are confirmed (CVE-2015-8106 and CVE-2015-8107) by the security community. The generated validators took additional 5-9% of the analysis time to check the correctness of analysis results.

## 1 Problem and Our Solution

Designing a correct static analysis and implementing it for a scalable analyzer in realistic setting have been two separate things. Designing a correct static analysis is exercising a theory such as abstract interpretation. Implementing the design into an executable analyzer that scales to realistic code size (million lines of code) is a different, onerous process that involves a lot of implementation engineering.

During the implementation phase, it is easy to have an analyzer that is not faithful to the design. Engineering in the implementation can unwittingly break the current design. Because achieving a scalable and precise static analyzer involves multiple implementations for design changes, the chance of unfaithful implementations is not low. When the analysis scalability or the precision is

low, we often have to redesign the analysis and change the implementation accordingly. After several iterations of this process can we arrive at an analyzer that are useful at least for a target set of code base. During this repetitive process it is not unusual that the correctness proof of the design becomes just a decorum rather than a trustworthy correctness guarantee of the analysis result of the final implementation.

We report a software framework, called ZooBerry, that fills the gap between static analysis designs (abstract semantics and their soundness proofs) and their faithful & scalable implementations (industrial-strength global analyzers whose results can be automatically checked correct). The input to ZooBerry is an abstract semantics (an analysis specification) and its soundness proof, both in Coq. For scalable implementation, ZooBerry automatically integrates into the abstract semantics the general sparse analysis technology [27,26] that streamlines both the spatial and temporal footprints of the generated analyzers. For faithful implementation, ZooBerry also generates a verified validator (and its safety proof in Coq) that will check the correctness of each analysis result of the generated sparse analyzer. We can continue to use this validator to assure that posterior changes for further cost reduction of the generated analyzers still conforms to the original, proven-correct design.

ZooBerry generates analyzers for an imperative language, into which C-like programs must first be translated. The generated analyzer is a global static analyzer that computes a table from the program points to abstract memories that safely summarize all possible memories that can occur at each program point.

The general sparse analysis techniques [27,26] that ZooBerry integrates into the generated analyzers is the optimizations that exploit the *spatial* and *temporal* sparsity of the semantics. For "spatial sparsity", the abstract memories entering to each program point is a sparse table whose entries are restricted to those to be accessed at the program point. For "temporal sparsity", the changed entries in the output memory after each program point is delivered directly to their use points. ZooBerry implements the sparse analysis framework [27,26] that guarantees the sparse version to preserve the precision of the given analysis, *i.e.,* $\mathit{fix}\hat{F} \approx \mathit{fix}\hat{F}_{\mathsf{sprs}}$ where $\hat{F}_{\mathsf{sprs}}$ is a sparse version of the abstract semantic function $\hat{F}$.

ZooBerry also generates a verified validator, with its correctness proof. The validator safely checks, for each analysis result from the generated analyzer, whether the result is a fixpoint of the designed abstract semantic function. This validator will be particularly useful in the following common situations in developing realistic analyzers. Once we have an automatically generated analyzers we sometimes manually engineer them further to squeeze additional performance gains. In this case too we can use the same validator to assure that the analysis results of such additionally tuned analyzers are faithful to the proven-correct design(specification).

Our following experience also motivates us to develop ZooBerry:

– Hard to implement: Implementing the sparse analysis is an onerous process. Sparse analyzer developers have to implement a pre-analysis, which gathers information about which abstract locations will be accessed by each program point, and its sparse data-dependence graph constructors. In our previous implementation [27], such additional code took about 2KLoC in OCaml, which accounted for 13% of the entire sparse analyzer code.
– Hard to debug: Debugging realistic static analyzers is hard because of its complicated abstract semantics and huge analysis results. In our previous work [15], using the formal verification technique, we found 13 bugs from our sparse analyzer implementation. The found bugs were so tricky that had not been detected in spite of extensive testings on large benchmark programs for years. Bug-free analyzers are especially difficult to develop because *(i)* tiny test cases cannot cover all corner cases of analyses, even though they were designed carefully; *(ii)* the analysis results of real-world software are too big to inspect manually for human; and *(iii)* even worse, as time goes on, many optimizations are applied to the analyzer and it becomes complicated more and more.

The ZooBerry framework is depicted in Fig. 1. The ZooBerry framework integrates the abstract semantics with other ready-made components for their faithful & sparse analyzers, which consist of pre-analyzers for sparsity exploitation and the worklist-based optimized fixpoint iteration engine.

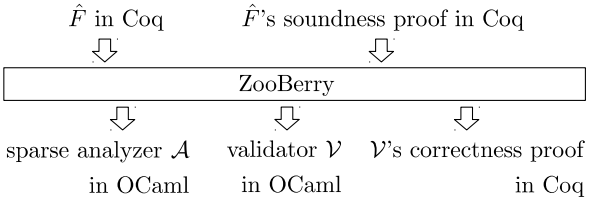


**Fig. 1.** Inputs and outputs of ZooBerry

ZooBerry's performance is tested in realistic settings. Using ZooBerry, we implemented two kinds of analyzers, an interval domain analyzer for buffer overrun bug detection and a taint analyzer for format string bug detection. At the moment of this writing, we could run the generated sparse interval (and taint) analyzers for up-to 98K (and 45K) lines of C programs. Also, with only 5-9% overhead to the sparse analysis time, it validated all of the analysis results successfully. Meanwhile, the implemented analyzers are practical to find real-world bugs. By using the ZooBerry-generated taint analyzer, together with our interactive false-alarm classification technology [16], we could identify security vulnerabilities in `latex2rtf` and `a2ps`, which were confirmed by the security community (two CVE numbers [20,21]).

In this work we make the following contributions.

– We report a software framework that fills the gap between static analysis designs and their faithful yet scalable implementations. The benefits of using the framework are:
  - Users can get global static analyzers that are scalable by the sparse analysis technique;
  - Users can get validators that check if the sparse analysis results for the input programs are sound approximations of the concrete semantics;
  - The correctness of the validator is automatically proven in Coq;
  - The framework provides proof libraries (boilerplate proofs) to minimize the proof parts expected from the users about the abstract semantics.
– We show this formally verified validation approach is useful in practice by implementing two sparse analyzers with ZooBerry and measuring the validation overhead.
– We share all of the source code of the ZooBerry framework and implemented analyzers on `http://ropas.snu.ac.kr/zooberry` .

*Organization.* Section 2 presents the input and the output of the framework. Section 3 and 4 show how to generate the analyzer and the validator. Section 5 discusses the correctness of the validator. Section 6 evaluates the performance of the framework with case studies, and Section 7 concludes with discussion of related works.

## 2 Overview of the ZooBerry Framework

### 2.1 Target Language

The target language of ZooBerry is a C-like imperative language that is represented as a control flow graph with commands in its nodes. Every node of the graph has a command such as pointer, array, and structure accesses. ZooBerry can analyze any languages that can be translated into the target language, including C using the CIL [24] frontend library.

$$
\begin{array}{rcl}
c & \rightarrow & lv \texttt{ = } e \mid lv \texttt{ = alloc}(e) \mid \texttt{assume}(e < e) \mid \texttt{call}(f, x, e) \mid \texttt{return}(e) \\
e & \rightarrow & n \mid e + e \mid lv \mid \texttt{\&}lv \\
lv & \rightarrow & x \mid \texttt{*}x \mid e\texttt{[}e\texttt{]} \mid e\texttt{.}x
\end{array}
$$

### 2.2 Input: Analysis Specification

ZooBerry users should give the analysis specification, the abstract semantics and its soundness proof in Coq. Users do not express anything about sparse analysis in the abstract semantics.

**Abstract Semantics** The ZooBerry framework assumes the basic structure of the abstract domain in order to enable general application of sparse optimization.

$$\hat{\mathbb{S}} \triangleq \text{Node} \to \hat{\mathbb{M}} \qquad\qquad \hat{\mathbb{M}} \triangleq \hat{\mathbb{L}} \to \hat{\mathbb{V}}$$

- An abstract state $\hat{s} \in \hat{\mathbb{S}}$ is a map from the node set (program points) to abstract memory.
- An abstract memory $\hat{m} \in \hat{\mathbb{M}}$ is a map from abstract location to abstract value.
- Then, users define their own domains of abstract locations $\hat{\mathbb{L}}$ and abstract values $\hat{\mathbb{V}}$.

For users to easily construct their own custom domains, ZooBerry provides primitive domains and domain constructors, such as the boolean, interval, sum, product, set, and map domains.

After defining the abstract domain, users should provide an abstract semantic function $\hat{f} : \mathbb{C} \times \hat{\mathbb{M}} \to \hat{\mathbb{M}}$ that calculates the next abstract memory when given a command and a memory. Here, in order for the framework to automatically generate sparse analyzer, users should use the provided memory management functions such as $\mathsf{lookup} : \hat{\mathbb{L}} \times \hat{\mathbb{M}} \to \hat{\mathbb{V}}$ and $\mathsf{update} : \hat{\mathbb{L}} \times \hat{\mathbb{V}} \times \hat{\mathbb{M}} \to \hat{\mathbb{M}}$ to access the memory domain $\hat{\mathbb{M}}$. See Sect. 3.2 and 5.1 for more details on this restriction.

**Soundness Proof** Users need to prove the soundness of the abstract semantic function defined in Coq, *i.e.,*

$$\forall c \in \mathbb{C}, m \in \mathbb{M}, \hat{m} \in \hat{\mathbb{M}}.\ m \in \gamma(\hat{m}) \Rightarrow f(c, m) \in \gamma(\hat{f}(c, \hat{m}))\ ,$$

where $c$ is a command and $m$ and $\hat{m}$ are respectively concrete and abstract memories.

Additionally, users need to prove that the abstract semantic function meets a necessary condition for sound sparse analysis: if an abstract memory $\hat{m}_2$ does not have memory entries that are accessed by $\hat{f}(c, \hat{m}_1)$, then the join with $\hat{m}_2$ commutes with the application of $\hat{f}$:

$$\forall c \in \mathbb{C}, m_1, m_2 \in \hat{\mathbb{M}}.\ \mathsf{disjoint}(\mathsf{acc}(\hat{f}, c, \hat{m}_1), \hat{m}_2) \Rightarrow \hat{f}(c, \hat{m}_1) \sqcup \hat{m}_2 = \hat{f}(c, \hat{m}_1 \sqcup \hat{m}_2)\ ,$$

where $\mathsf{acc}$ collects abstract locations accessed during analysis (Def. 4) and $\mathsf{disjoint}$ denotes that a set of abstract locations are disjoint to memory entries that are bound to non-bottom values (Def. 5). Note that $\mathsf{acc}$ can be easily defined thanks to the restriction that users should use pre-defined memory access functions such as $\mathsf{lookup}$ and $\mathsf{update}$. This condition is easily provable thanks to a set of static-analysis-specific boilerplate lemmas and proof automation tools in Coq provided in ZooBerry.

### 2.3 Output: Sparse Analyzer and Verified Validator

Given an analysis specification described in the previous sections, ZooBerry generates an executable sparse analyzer implemented in OCaml (Sect. 3), and a

validator that is formally verified in Coq (Sect. 4 and 5). The proven validator is extracted to OCaml and then executed.

Fig. 2 shows how an input program is analyzed and then validated in the ZooBerry framework (for now, please ignore the densifier). The generated analyzer is given an input program, and the generated validator is given the input program and its sparse analysis result. If the result is unsuccessfully validated, then it is due to one of the three reasons: *(i)* user-defined analysis specification is unsound; *(ii)* sparse analysis components, *e.g.,* data dependency graph generator or fixpoint iterator, have bugs; or *(iii)* the validator is incomplete, *i.e.,* it cannot validate all the correct sparse analysis results. Case *(i)* is avoided when users give a soundness proof of the abstract semantics. Regarding cases *(ii)* and *(iii)*, ZooBerry has so far been stabilized by debugging validation failures during the development, and now we can hardly observe the bugs because of *(ii)* and *(iii)*.

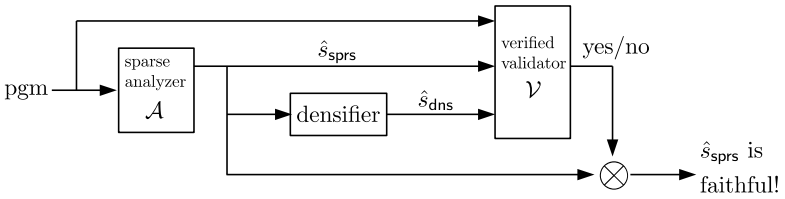


**Fig. 2.** Validation process of sparse analysis results.

Provided that users give the soundness proof and the analysis specification, ZooBerry formally guarantees the correctness of the generated validator: if the sparse analysis result $\hat{s}_{\mathsf{sprs}}$ is successfully validated, then $\hat{s}_{\mathsf{sprs}}$ is a sound approximation of the concrete semantics of the input program.

### 2.4 Components of ZooBerry

ZooBerry contains a sparse analyzer and a verified validator that are parameterized by user inputs, which are abstract semantics and its soundness proof, as depicted in Fig. 3. Except for the user inputs (dashed boxes), the others (solid boxes) are pre-built in ZooBerry. Most of the pre-built parts except for the concrete semantics depend on the user inputs. For example, the pre-/main-analyzers can run only when the abstract semantics is provided and the correctness proof of validator is completed when the soundness proof of the abstract semantics is provided. The user-provided abstract semantics is shared by the sparse analyzer and the verified validator, but the soundness proof is only used by the latter.

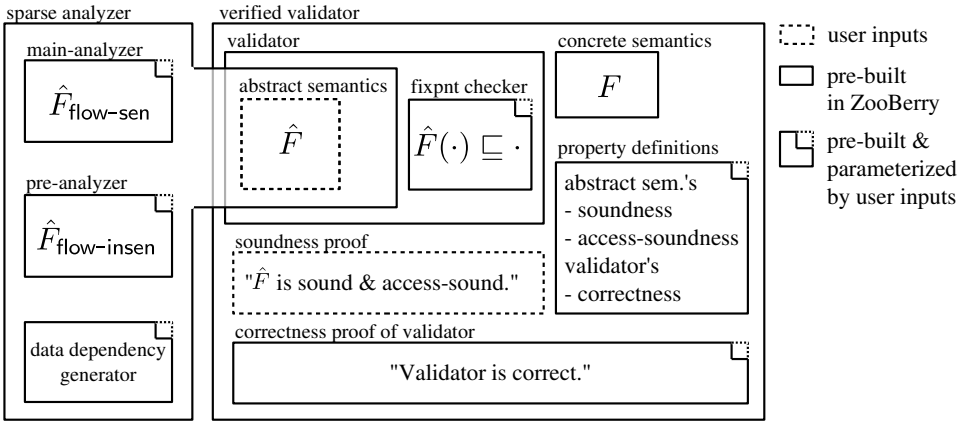


**Fig. 3.** Components of ZooBerry

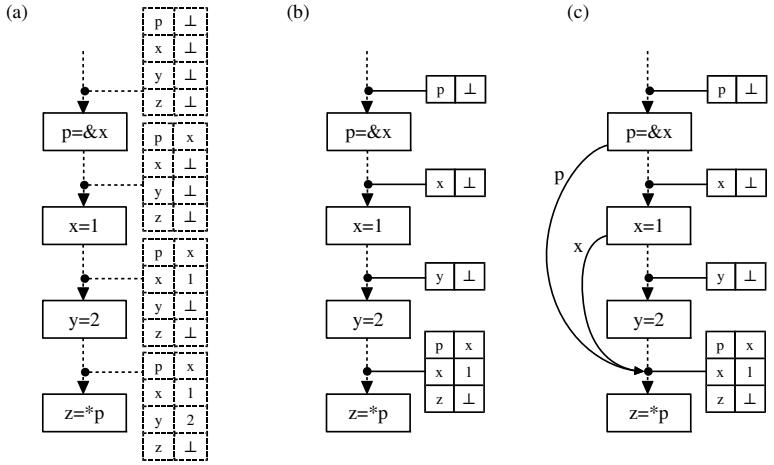


**Fig. 4.** An example of sparse analysis: (a) non-sparse analysis; (b) spatial sparsity; (c) sparse analysis with spatial and temporal sparsities.

## 3 Generation of Sparse Analyzer

### 3.1 Sparse Analysis

Sparse analysis [27,26] is an optimization technique that reduces analysis costs by passing only the updated memory portion directly from their def-points (where some abstract values are defined) to their next use-points (where they are used). Figure 4 shows an example of the sparse analysis optimization. In the non-sparse analysis, every abstract memories has all abstract locations as entries inefficiently. By the spatial sparsity, only accessed entries remain in the input memories. By the temporal sparsity, the values of the accessed entries are delivered directly from their def-points to use-points.

### 3.2 Sparse Analysis Implementation in ZooBerry

The sparse analysis in ZooBerry-generated analyzer is performed in the four steps:

Step 1. **Pre-analysis**: We have to collect information for sparse analysis. It performs flow-insensitive analysis to get a conservative analysis result.
Step 2. **Calculating access information**: From the pre-analysis result, it calculates a conservative *access information* about which abstract locations can be defined or used at each program points.
Step 3. **Drawing data dependency graph**: It draws a data dependency graph using the calculated access information in Step 2. For that, it uses a conventional SSA transformation algorithm [11] because drawing dependency edges are similar to matching every use of variables to static assignments in the SSA transformation.
Step 4. **Main analysis**: It performs flow-sensitive analysis using the data dependency graph drawn in Step 3. The information from Step 2 and 3 is used for spatial and temporal locality. During the analysis only those accessed memory portions (spatial sparsity) flow around the data-dependence edges (temporal sparsity).

**Calculating Access Information** Calculating access information of an arbitrary abstract semantics is almost impossible. For example, if users define and use their own abstract memory domains and functions modifying values of them, it is hard to observe which abstract locations are accessed.

In order to avoid such too-general situations, ZooBerry restricts the abstract memory domain and associated functions as briefly mentioned in Sect. 2.2.

- Users define their own abstract location and abstract memory domains, then the abstract memory domain is constructed by ZooBerry as a map from the former to the latter.
- Users should use given `lookup` or `update` functions by ZooBerry to manipulate abstract memories in their abstract semantic function.

With the above restriction, the ZooBerry system can control them to collect accessed abstract locations during analysis. For example, memory access functions, $\mathsf{lookup}'$, $\mathsf{update}'$, and $\mathsf{wupdate}'$ are defined below and they are instrumented to collect accessed abstract locations—in which, $\mathsf{wupdate}$ is a weak-update function. By the instrumentation, they can collect used and defined abstract locations into global variables *use* and *def*, respectively.

$$\mathsf{lookup}(\hat{l}, \hat{m}) \triangleq \hat{m}[\hat{l}] \qquad \mathsf{lookup}'(\hat{l}, \hat{m}) \triangleq \mathit{use} \leftarrow \mathit{use} \cup \{\hat{l}\} \;;\; \hat{m}[\hat{l}]$$

$$\mathsf{update}(\hat{l}, \hat{v}, \hat{m}) \triangleq \hat{m}[\hat{l} \mapsto \hat{v}] \qquad \mathsf{update}'(\hat{l}, \hat{v}, \hat{m}) \triangleq \mathit{def} \leftarrow \mathit{def} \cup \{\hat{l}\} \;;\; \hat{m}[\hat{l} \mapsto \hat{v}]$$

$$\mathsf{wupdate}(\hat{l}, \hat{v}, \hat{m}) \triangleq \hat{m}[\hat{l} \mapsto \hat{m}[\hat{l}] \sqcup \hat{v}]$$

$$\mathsf{wupdate}'(\hat{l}, \hat{v}, \hat{m}) \triangleq \mathit{def} \leftarrow \mathit{def} \cup \{\hat{l}\} \;;\; \mathit{use} \leftarrow \mathit{use} \cup \{\hat{l}\} \;;\; \hat{m}[\hat{l} \mapsto \hat{m}[\hat{l}] \sqcup \hat{v}]$$

With the instrumented memory access functions and the pre-analysis result, ZooBerry can calculate accessed locations at each program node: *(i)* the $\mathsf{get\_access}$ function first initializes the global variables of *def* and *use*; *(ii)* it applies the modified abstract semantic function $\hat{f}'$ using instrumented memory access functions with the pre-analysis result $\hat{m}_{\mathsf{pre}}$; then *(iii)* finally, it returns *def* and *use* in which accessed abstract locations are collected.

$$\mathsf{get\_access} : \mathbb{C} \times \hat{\mathbb{M}} \to 2^{\hat{\mathbb{L}}} \times 2^{\hat{\mathbb{L}}}$$

$$\mathsf{get\_access}(c, \hat{m}_{\mathsf{pre}}) \triangleq \mathit{def} \leftarrow \phi \;;\; \mathit{use} \leftarrow \phi \;;\; \hat{f}'(c, \hat{m}_{\mathsf{pre}}) \;;\; (\mathit{def}, \mathit{use})$$

Note that, in ZooBerry's implementation, the instrumented memory access functions are written with a monad type, just because Coq is pure functional and does not have reference.

**Monotonicity of Abstract Semantic Function** In order for a data dependency graph to be conservatively drawn before the main analysis, *i.e.,* not to miss data dependency of the main analysis, a user-defined abstract semantic function $\hat{f}$ should be monotone not only in return values but also in its memory accesses.

$$\forall c, \hat{m}_1, \hat{m}_2.\ \hat{m}_1 \sqsubseteq \hat{m}_2 \Rightarrow \hat{f}(c, \hat{m}_1) \sqsubseteq \hat{f}(c, \hat{m}_2) \wedge \mathsf{get\_access}(c, \hat{m}_1) \sqsubseteq \mathsf{get\_access}(c, \hat{m}_2)$$

Suppose there is an abstract semantic function $\hat{f}$ and an abstract memory $\hat{m}$ at a program point with a command $c$ during the main analysis. At this time, we want to be sure that $\mathsf{get\_access}(c, \hat{m}) \sqsubseteq \mathsf{get\_access}(c, \hat{m}_{\mathsf{pre}})$, otherwise, it may omit to draw some data dependency edges. It is easy to show the condition is satisfied if $\hat{f}$ is monotone w.r.t. both its return values and memory accesses.

- Since $\hat{f}$ is monotone w.r.t. its return values and more imprecise flow-insensitive analysis is used as the pre-analysis, it is guaranteed that $\hat{m}_{\mathsf{pre}}$ is bigger than $\hat{m}$, *i.e.,* $\hat{m} \sqsubseteq \hat{m}_{\mathsf{pre}}$.
- Thus, from the fact of $\hat{m} \sqsubseteq \hat{m}_{\mathsf{pre}}$ and $\hat{f}$'s monotonicity, we can conclude $\mathsf{get\_access}(c, \hat{m}_{\mathsf{pre}})$ is bigger than $\mathsf{get\_access}(c, \hat{m})$.

If the user-defined abstract semantics does not satisfies the monotonicities, validations may fail because the data dependency graph may not be correct.

**Safe Approximation of Access Information** ZooBerry-generated sparse analyzers meet the safe approximation conditions of access information in the sparse analysis framework [27, Lemma 2 on p. 4], if the user-defined $\hat{f}$ satisfies the above monotonicities.[1]

## 4 Validation of Sparse Analysis Results

The validation is a fixpoint check. When an analysis result $\hat{s}$ is a post-fixpoint of the sound abstract semantic function $\hat{F}$, we can say it soundly approximates concrete semantics by the abstract interpretation framework [7,8]. The validator $\mathcal{V}$ can be defined by,

$$\mathcal{V}(p, \hat{s}) \triangleq \hat{F}_p(\hat{s}) \sqsubseteq \hat{s}$$

However, the above validation is not that straightforward because of the sparsity. For example, at the first node in Fig. 4, the input memory is $\{p \mapsto \bot\}$ and the memory at its output position is $\{x \mapsto \bot\}$. This sparse result cannot be a post-fixpoint of $\hat{f}$; by the spatial sparsity, the input memory of the second does not have an entry for p.

$$\hat{f}(\text{p=\&x}, \{p \mapsto \bot\}) \not\sqsubseteq \{x \mapsto \bot\}$$

We solved this problem by: *(i)* recovering a dense result from the sparse result; then *(ii)* performing the simple-minded fixpoint check to the densified one. We elaborate the validation method in the following sections.

### 4.1 Correctness of Validator

In this section, we redefine the validator and its correctness for the context of sparse analysis. There are mainly two revision points.

- The validator conducts the fixpoint check on the densified result, not on the sparse one—as of now, suppose there exists a densifier that recovers densified analysis results from sparse ones, which will be addressed thoroughly in the next sections.
- The validator checks the *consistency* between the sparse and densified results. By consistency we mean non-bottom values in the sparse results should coincide with those in the densified one. The consistency ensures that the sparse result is meaningful: if it has a value for a location, it actually over-approximates the concrete value for the location.

[1] In the condition [27, Lemma 2 on p. 4], "$\mathsf{D}(c)$" and "$\mathsf{U}(c)$" are respectively the sets of defined and used abstract locations when analyzing command $c$. "$\hat{\mathsf{D}}(c)$" and "$\hat{\mathsf{U}}(c)$" are their approximations. In ZooBerry's implementation, they are as follows:

$$\mathsf{D}(c) = \mathtt{fst}(\mathtt{get\_access}(c, \hat{m})) \qquad \hat{\mathsf{D}}(c) = \mathtt{fst}(\mathtt{get\_access}(c, \hat{m}_{\mathsf{pre}}))$$
$$\mathsf{U}(c) = \mathtt{snd}(\mathtt{get\_access}(c, \hat{m})) \qquad \hat{\mathsf{U}}(c) = \mathtt{snd}(\mathtt{get\_access}(c, \hat{m}_{\mathsf{pre}}))$$

where $\hat{m}$ and $\hat{m}_{\mathsf{pre}}$ are abstract memories appearing in the main and pre analyses.

$$\mathcal{V}(p, \hat{s}_{\mathsf{sprs}}, \hat{s}_{\mathsf{dns}}) \triangleq \hat{F}_p(\hat{s}_{\mathsf{dns}}) \sqsubseteq \hat{s}_{\mathsf{dns}} \text{ and } \hat{s}_{\mathsf{sprs}} \sim \hat{s}_{\mathsf{dns}}$$
$$\hat{s}_{\mathsf{sprs}} \sim \hat{s}_{\mathsf{dns}} \triangleq \forall n \in \text{Node}, \hat{l} \in \hat{\mathbb{L}}.\ \hat{s}_{\mathsf{sprs}}(n, \hat{l}) = \bot \ \vee\ \hat{s}_{\mathsf{sprs}}(n, \hat{l}) = \hat{s}_{\mathsf{dns}}(n, \hat{l}) \ ,$$

where $\hat{s}_{\mathsf{sprs}}$ and $\hat{s}_{\mathsf{dns}}$ are respectively sparse and densified analysis results.

The correctness of validator should also take into account the consistency between the sparse and densified states. For a correct validator, a validation success implies the soundness of the non-bottom values in the sparse state.

**Definition 1 (Correctness of validator).** *A validator $\mathcal{V}$ is correct if successful validations imply that the validated analysis result $\hat{s}_{\mathsf{sprs}}$ is a sound approximation of concrete semantics $[\![p]\!]$ with regard to non-bottom values in it,*

$$\forall p, \hat{s}_{\mathsf{sprs}}, \hat{s}_{\mathsf{dns}}.\ \mathcal{V}(p, \hat{s}_{\mathsf{sprs}}, \hat{s}_{\mathsf{dns}}) = \mathsf{true} \Rightarrow [\![p]\!] \in \gamma(\hat{s}_{\mathsf{dns}}) \ \wedge\ \hat{s}_{\mathsf{sprs}} \sim \hat{s}_{\mathsf{dns}} \ .$$

### 4.2 Densification

For validation successes, a densifier that recovers $\hat{s}_{\mathsf{dns}}$ from $\hat{s}_{\mathsf{sprs}}$ should satisfy two conditions: *(i)* the densified abstract state should be a post-fixpoint of the non-sparse abstract semantic function, *i.e.,* $\hat{F}(\hat{s}_{\mathsf{dns}}) \sqsubseteq \hat{s}_{\mathsf{dns}}$; *(ii)* densification must not change non-bottom values from the sparse analysis result, $\hat{s}_{\mathsf{sprs}} \sim \hat{s}_{\mathsf{dns}}$.

A naive approach is repeatedly applying the non-sparse abstract semantic function $\hat{F}$ until the abstract state becomes a post-fixpoint, but it is too expensive. In our early experiments, it took much more time than the current implementation, because it required many comparisons of abstract states for fixpoint checking, which nullified the gains from the sparse analysis optimization.

In order to efficiently densify the analysis result, we devise a densification algorithm that visits every node only once by exploiting data dependency graphs.

**Observation** Suppose an abstract semantic function at node $n$ does not use an abstract location $\hat{l}$ and the value of $\hat{l}$ is defined at some ancestor nodes of $n$. Note that this is the only case that requires densification, *i.e.,* if the value of $\hat{l}$ is used at $n$ already or is never defined at any ancestors of $n$, then $n$ does not need to have the value of $\hat{l}$.

- When the value of $\hat{l}$ is not defined between $n$ and its immediate dominator, $\mathsf{idom}(n)$: If we somehow densifies the abstract memory of $\mathsf{idom}(n)$ first, then the value of $\hat{l}$ at $\mathsf{idom}(n)$ can be used to densify the value of $\hat{l}$ at $n$, because the value should not change between the $n$ node and its immediate dominator.
- When the value of $\hat{l}$ is defined between $n$ and its immediate dominator, $\mathsf{idom}(n)$: In this case, the value of $\hat{l}$ should have been written at the $n$ node because our algorithm drawing data dependency graphs mimics the SSA transformation [11]. If a value is defined between $n$ and $\mathsf{idom}(n)$, the $n$ node should be a phi node in the SSA transformation. Thus, our algorithm considers the phi node as both a use-point and a def-point of the value and draws dependency edges for them, though an actual phi command is not added to the node.

According to the observation, we design a simple efficient densification algorithm, which is presented in Fig. 5. During the densification, *(i)* all of the nodes are visited once in a domination order, thus, when visiting a node $n$, assume that the immediate dominator of $n$ was densified before; *(ii)* if the node $n$ has the value of $\hat{l}$, *i.e.,* $\hat{l} \neq \bot$, the densifier leaves it unchanged, otherwise, the value of $\hat{l}$ is densified with that of the immediate dominator of $n$.

**Require:** $\hat{s}_{\mathsf{sprs}}$ (sparse abstract state), $\mathsf{dom\_tree}$ (dominator tree of program CFG)
**Ensure:** $\hat{s}_{\mathsf{dns}}$ (densified abstract state)

1: $\hat{s} \leftarrow \hat{s}_{\mathsf{sprs}}$
2: **for all** $n \in \mathsf{dom\_tree}$ in BFS order **do**
3: **if** $n \neq \mathsf{root}$ **then**
4: $\hat{m} \leftarrow \hat{s}(n), \quad \hat{m}_{\mathsf{idom}} \leftarrow \hat{s}(\mathsf{immediate\_dominator\_of}(n))$
5: $\hat{s}(n) \leftarrow \{\hat{l} \mapsto \hat{v} \mid (\hat{m}[\hat{l}] \neq \bot \wedge \hat{v} = \hat{m}[\hat{l}]) \vee (\hat{m}[\hat{l}] = \bot \wedge \hat{v} = \hat{m}_{\mathsf{idom}}[\hat{l}])\}$
6: $\hat{s}_{\mathsf{dns}} \leftarrow \hat{s}$

**Fig. 5.** Densification algorithm

Note that the densification algorithm is tightly coupled with the algorithm drawing data dependency graphs in Sect. 3.2. If the drawing algorithm changes, the densifier should also change, otherwise validations would fail with incorrectly densified states.

On the other hand, the densifier and the data dependency drawing algorithm are not in the trust bases in the ZooBerry system. Namely, the correctness of the validator does not depend on them as shown in Def. 1, instead the validator checks whether or not the densified one is correct afterwards. Therefore, even if we change the data dependency drawing algorithm and the densifier accordingly, the correctness of proof of the validator remains intact.

## 5 Formal Correctness Proof of Validator

In Coq, we formally proved that the validator is correct (Def. 1), if the correctness proofs of user-defined abstract semantic function are given. In this section, we define the two correctness requirements, *soundness* (Def. 2) and *access-soundness* (Def. 3), that a user-defined abstract semantics function should satisfy in ZooBerry.

**Theorem 1.** *If a user-defined abstract semantic function $\hat{f}$ is sound and access-sound, the validator $\mathcal{V}_{\hat{f}}$ instantiated with $\hat{f}$ is correct.*

### 5.1 Proofs of Abstract Semantic Function

The definition of the soundness is conventional in that the abstract semantic function should approximates the pre-defined concrete semantics. The abstraction is represented by only the concretization function $\gamma$ because sometimes

the abstraction function $\alpha$ is non-constructive, which is hard to deal with in Coq [29]. The abstraction should be written by users because users define their own abstract domains and the abstraction cannot be pre-defined by ZooBerry.

**Definition 2 (Soundness).** *An abstract semantic function $\hat{f}$ is sound if it approximates concrete semantics soundly.*

$$\forall c, m, \hat{m}.\ m \in \gamma(\hat{m}) \Rightarrow f(c, m) \in \gamma(\hat{f}(c, \hat{m}))$$

In addition to the soundness, users should prove another condition, called *access-soundness*, which says that an abstract semantic function never influences or be influenced by *non-accessed* abstract values. When analyzing a command $c$ in sparse analysis, an input abstract memory is divided into two pieces: one has memory entries that is known to be accessed by pre-analysis ($\hat{m}_1$) and the other has disjoint memory entries known to be not accessed ($\hat{m}_2$). In a usual analysis, they are all given for analysis, *i.e.,* $\hat{f}(c, \hat{m}_1 \sqcup \hat{m}_2)$. On the other hand, in sparse analysis, only the former one, $\hat{m}_1$, is given like $\hat{f}(c, \hat{m}_1)$. In order to prove that the sparse result is the same as the non-sparse case with regard to its accessed memory entries, it is sufficient to show that $\hat{f}(c, \hat{m}_1) \sqcup \hat{m}_2 = \hat{f}(c, \hat{m}_1 \sqcup \hat{m}_2)$. The condition dictates that the sparse result does not affect $\hat{m}_2$ and also is not affected by $\hat{m}_2$.

**Definition 3 (Access-soundness).** *Suppose a set of accessed abstract locations are $\hat{ls}$ when applying an abstract semantic function $\hat{f}$ to an abstract memory $\hat{m}_1$. The $\hat{f}$ function is access-sound if every application of $\hat{f}$ influences and is influenced by only abstract values associated with $\hat{ls}$.*

$$\forall c, m_1, m_2.\ \mathsf{disjoint}(\mathsf{acc}(\hat{f}, c, \hat{m}_1), \hat{m}_2) \Rightarrow \hat{f}(c, \hat{m}_1) \sqcup \hat{m}_2 = \hat{f}(c, \hat{m}_1 \sqcup \hat{m}_2)$$

**Definition 4 (acc function).** *Given an abstract semantic function $\hat{f}$, a command $c$, and an abstract memory $\hat{m}$, the function* acc *returns a set of abstract locations accessed during the function application of $\hat{f}$ to $(c, \hat{m})$.*

$$\mathsf{acc}(\hat{f}, c, \hat{m}) \triangleq \text{let } (\mathit{def}, \mathit{use}) = \mathsf{get\_access}_{\hat{f}}(c, \hat{m}) \text{ in}$$
$$\mathit{def} \cup \mathit{use}$$

**Definition 5 (Disjoint).** *An abstract memory $\hat{m}$ and a set of abstract locations $\hat{ls} \in 2^{\hat{\mathbb{L}}}$ are disjoint if $\hat{m}$ does not include a mapping from an abstract location $\hat{l}$ in $\hat{ls}$ into a non-bottom value, i.e., all of the images of $\hat{ls}$ in $\hat{m}$ are the bottom value.*

$$\mathsf{disjoint}(\hat{ls}, \hat{m}) \triangleq \forall \hat{l} \in \hat{ls}.\ \hat{m}[\hat{l}] = \bot$$

Note that the property is relatively easy to prove, because it is compositional in the sense that if functions $f_1$ and $f_2$ ($\in \hat{M} \to \hat{M}$) is access-sound, then their composition $f_1 \circ f_2$ is also access-sound. Therefore, users just can decompose the abstract semantic function into subfunctions, then prove each of them is access-sound.

### 5.2 Trusted Computing Base

The trusted computing base (TCB) of ZooBerry contains the Coq proof checker, the OCaml code extractor translating Coq code to OCaml code, the OCaml runtime, and frontend such as the CIL parser and a translation function from input programs in CIL to ZooBerry's target language.

The sparse analyzer, the densifier, and the validator are not included in the TCB. The generated validator in OCaml is not TCB because it is extracted from a validator in Coq, the correctness of which is completed and checked by Coq from the user-provided correctness proof of the abstract semantics. The generated sparse analyzer and the densifier are not TCB because their outputs do not affect the correctness of the final faithful result as shown in Fig. 2. Note that the faithful $\hat{s}_{\mathsf{sprs}}$ is available only when the validation is correct.

## 6 Evaluation

We evaluate the performance of ZooBerry with two cases of using ZooBerry: an interval domain analyzer for buffer overrun bug detection and a taint analyzer for format string bug detection. We tested these two sparse analyzers for C benchmark program on a Linux machine with Intel i7 3.2GHz CPU and 24GB RAM.

**Scalability** Table 1 and 2 show the analysis cost of the ZooBerry-generated interval and taint analyzers. In order to make this scalability, many other optimizations in addition to the sparse analysis are integrated automatically by ZooBerry, for example, efficient worklist/widening strategies [3], selective memory operators [1], and access-based localization [28,25].

In general, taint analysis is known to be faster than interval analysis, but it was not in our implementation. Our taint analyzer needs more memory resources than the interval analyzer, because it collects taint source points as a set to make more informative alarm report, *i.e.,* the abstract taint domain is not the simple boolean domain. It is why the benchmark programs listed in Table. 1 and 2 are different; in the experiment tables, we lists the benchmark programs that were able to analyze within 24GB RAM memory resource.

**Validation Overhead** We found that the validations take only about 5-9% more time to the sparse analysis time in average.

**Usefulness** The analyzers generated by ZooBerry were useful to find real-world bugs in practice. For example, by using the ZooBerry-generated taint analyzer, together with our interactive false-alarm classification technology [16], we could identify 30 true format string vulnerabilities and two of them, from `latex2rtf` and `a2ps`, were confirmed by the security community receiving two CVE numbers [20,21] by their significance.

| Program | LoC | Total | Anal | Vali | Dens | Fixpt | Overhead(Vali) |
|---|---|---|---|---|---|---|---|
| gnuplot-4.2.6 | 78K | 26,077 | 25,212 | 864 | 314 | 550 | +3.4% |
| lout-3.38 | 34K | 20,034 | 19,571 | 463 | 173 | 289 | +2.3% |
| guile-1.0 | 44K | 12,225 | 11,995 | 230 | 155 | 75 | +1.9% |
| raptor-1.4.21 | 57K | 8,346 | 8,216 | 129 | 69 | 60 | +1.5% |
| latex2rtf-2.3.8 | 20K | 3,288 | 3,120 | 168 | 110 | 57 | +5.3% |
| gcal-3.6.3 | 98K | 1,498 | 1,444 | 54 | 20 | 34 | +3.7% |
| bison-2.5 | 54K | 1,413 | 1,361 | 52 | 22 | 29 | +3.8% |
| urjtag-r1476 | 42K | 1,274 | 1,163 | 110 | 33 | 77 | +9.5% |
| sextractor-2.4.4 | 22K | 1,168 | 1,154 | 14 | 6 | 7 | +1.2% |
| wget-1.9 | 23K | 547 | 520 | 27 | 12 | 15 | +5.2% |
| a2ps-4.14 | 40K | 472 | 450 | 22 | 10 | 11 | +4.9% |
| lsh-2.0.4 | 68K | 431 | 409 | 21 | 13 | 8 | +5.3% |
| icecast-server-1.3.12 | 18K | 374 | 353 | 21 | 11 | 9 | +6.0% |
| tree-puzzle-5.2 | 45K | 204 | 195 | 8 | 3 | 4 | +4.4% |
| uucp-1.07 | 53K | 125 | 117 | 8 | 4 | 4 | +6.8% |
| pies-1.2 | 46K | 115 | 108 | 7 | 3 | 3 | +6.7% |
| sed-4.0.8 | 18K | 94 | 89 | 5 | 2 | 2 | +6.1% |
| mpg123-1.12.1 | 28K | 87 | 79 | 8 | 2 | 5 | +10.1% |
| patch-2.7 | 35K | 86 | 78 | 7 | 2 | 4 | +9.1% |
| daemon-0.6.4 | 28K | 58 | 51 | 7 | 3 | 3 | +14.0% |
| diffutils-3.2 | 60K | 41 | 38 | 3 | 1 | 1 | +9.1% |
| enscript-1.6.5 | 38K | 40 | 39 | 1 | 0.7 | 0.6 | +3.2% |
| | | | | | | **Average** | +5.6% |

**Table 1.** Benchmark test of an interval analyzer generated by ZooBerry. All of the validations succeeded. **LoC** denotes sizes of input programs without comments and empty lines. **Anal** and **Vali** denote time, in seconds, spent for sparse analysis and validation, respectively. The validation time is divided into two phases, densification (**Dens**) and fixpoint check (**Fixpt**). **Overhead** denotes additional validation costs compared to the sparse analysis cost, *i.e.,* $\frac{\textbf{Vali}}{\textbf{Anal}} \times 100$.

| Program | LoC | Total | Anal | Vali | Dens | Fixpt | Overhead(Vali) |
|---|---|---|---|---|---|---|---|
| icecast-server-1.3.12 | 18K | 992 | 904 | 87 | 44 | 42 | +9.6% |
| enscript-1.6.5 | 38K | 390 | 381 | 9 | 3 | 5 | +2.3% |
| tree-puzzle-5.2 | 45K | 291 | 282 | 8 | 3 | 4 | +3.0% |
| rnv-1.7.10 | 6K | 220 | 209 | 11 | 6 | 4 | +5.2% |
| sed-4.0.8 | 18K | 132 | 126 | 5 | 2 | 2 | +4.3% |
| bc-1.06 | 10K | 128 | 119 | 9 | 4 | 5 | +7.9% |
| aalib-1.4p5 | 10K | 94 | 92 | 1 | 0.8 | 0.5 | +1.4% |
| gnuchess-5.05 | 8K | 37 | 34 | 2 | 1 | 1 | +8.1% |
| agedu-8642 | 3K | 37 | 34 | 3 | 1 | 2 | +9.2% |
| unrtf-0.19.3 | 5K | 22 | 19 | 2 | 1 | 1 | +15.1% |
| gzip-1.2.4a | 5K | 20 | 18 | 1 | 0.7 | 0.6 | +6.9% |
| e2ps-4.34 | 6K | 4 | 3 | 0.5 | 0.2 | 0.3 | +13.5% |
| gbsplay-0.0.91 | 5K | 4 | 3 | 0.8 | 0.4 | 0.4 | +22.8% |
| archimedes-0.7.0 | 4K | 8 | 7 | 0.7 | 0.4 | 0.3 | +9.4% |
| | | | | | | **Average** | +8.5% |

**Table 2.** Benchmark test of a taint analyzer generated by ZooBerry. All of the validations succeeded. The experiment setting is the same to that of the interval analyzer in Table. 1.

**Implementation Costs** In our two analyzer implementations, analysis specification and its soundness proof account for 17-19% of the entire code for each analyzer as shown in Table. 3. It took about 3.5 man-months to define the abstract semantics of the interval analyzer and its correctness proof. On the other hand, the taint analyzer and its proof were developed within two weeks. It was quick because we did not have to write them from scratch: we copied and modified the user inputs of the interval analyzer to deal with taint information, therefore some parts of them were used with just small modifications, *e.g.,* a specification and proof on points-to information.

| | ZooBerry | | Intrvl Anal | | Tnt Anal | |
|---|---|---|---|---|---|---|
| | Coq | OCaml | Spec | Proof | Spec | Proof |
| **LoC** | 5,042 | 2,523 | 689 | 854 | 766 | 945 |
| **Total** | 7,565 | | 1,543 | | 1,711 | |

**Table 3.** Code size of ZooBerry. **LoC** denotes sizes of code without comments and empty lines. **Spec** and **Proof** denote user-defined analysis specifications and soundness proofs of them, respectively.

**Dealing with External Function** When some part of source code is not given (such as for external library functions), users can define abstract semantics for them and use the semantics in their analyzers and validators. When an analyzer encounters a call statement to an external library function, it uses the abstract semantics defined by users. If there is no available abstract semantics, the call

statement is considered a nop statement. Note that the validator uses the same abstract semantics (nop) for the external function calls.

Even if there are some abstract semantics on external functions, users can formally prove the other parts of their abstract semantics sound. In ZooBerry's concrete semantics, external function calls are not defined, *i.e.,* there is no small-step relation on them. Therefore, when proving soundness, users do not need to prove on the external function calls.

## 7 Discussion

ZooBerry is the first system of its kind in two aspects: for C-like languages it generates 1) global semantic-based static analyzers that are tuned for scalable realistic analyses and 2) their verified validators that check the correctness of the analysis results. The generated global analyzers are equipped with the general sparse techniques [28,25,27,26] that exploit both the spatial and temporal sparsities of the abstract semantics (analysis specification) while preserving the analysis precision.

Other systems [17,5,19,2,14] focus on verified analyzers. The scalability of those verified analyzers is less emphasized than ZooBerry. In those works, test programs to analyze did not exceed 3KLoC [5,19,2,14] or sometimes the scalability was not even mentioned [17]. They focus on achieving verified analyzers. ZooBerry focuses on achieving not only scalable (by the sparse analysis framework) but also checkable analyzers (by the verified validator approach).

The main difference from Verasco [14] is the verification targets; while ZooBerry verifies the validator, Verasco verifies the analyzer directly. The verified validator approach has pros and cons. It is much cheaper than verifying static analyzers directly. On the other hand, the validation has runtime overhead because it should run every time analyzers return results. To decrease the runtime overhead, we carefully designed the validator and made it has only 5-9% over the analysis time.

Soot [18], a software framework for static analyzer generator for Java-family languages, does not generate a verified validator. The framework is also less general than ZooBerry. Soot focuses more on simple, data-flow-analysis-like static analyses than arbitrary abstract semantics. The generated analyzers does not integrated the general sparse techniques that ZooBerry does.

Infer [12,6] from Facebook is a software framework to compose static analyzers that are scalable by mean of Infer's modular analysis platform. Infer's modular platform is not general enough to support arbitrary abstract semantics. Infer does not provide any tool to verify the analysis results.

Frama-C [9] does not support formal verification to make the user-provided parts of the final analyzers faithful.

ZooBerry-generated validators perform a kind of the translation validation [30]. ZooBerry-generated analyzers translate input programs into their abstract semantics. The validator checks whether this translation is correct. The property to check is whether the translation preserves an abstraction relation between

concrete semantics of a program and its approximation, rather than semantic equivalence between programs as in compilers. ZooBerry-generated sparse analyzers and their validators can also be seen in the proof-carrying code [23] context. ZooBerry-generated sparse analyzers produce as small proofs (analysis results) as possible by the sparse technique. Such "compressed" proofs are carried to ZooBerry-generated validators, where the proofs are checked after a decompression (densification, Sect. 4.2).

## References


1. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 196–207, New York, NY, USA, 2003. ACM.
2. Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal Verification of a C Value Analysis Based on Abstract Interpretation. In Manuel Fahndrich and Francesco Logozzo, editors, *SAS - 20th Static Analysis Symposium*, volume Lecture Notes in Computer Science of *7935*, pages 324–344, Seattle, United States, June 2013. Springer.
3. Franois Bourdoncle. Efficient chaotic iteration strategies with widenings. In *In Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993.
4. David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. In *Proc. of 13th European Symposium on Programming (ESOP'04)*, number 2986 in Lecture Notes in Computer Science, pages 385–400. Springer-Verlag, 2004.
5. David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. *Theoretical Computer Science*, 342(1):56–78, September 2005. Extended version of [4].
6. Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
8. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.
9. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.
10. Pascal Cuoq, Benjamin Monate, Anne Pacalet, and Virgile Prevosto. Functional dependencies of c functions via weakest pre-conditions. *Int. J. Softw. Tools Technol. Transf.*, 13(5):405–417, October 2011.

11. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.
12. Facebook. *Infer: A static analyzer for Java, C, C++, and Objective-C*. `https://github.com/facebook/infer`.
13. Alexis Fouilhe and Sylvain Boulmé. A certifying frontend for (sub)polyhedral abstract domains. working paper or preprint, May 2014.
14. Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified c static analyzer. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 247–259, New York, NY, USA, 2015. ACM.
15. Jeehoon Kang, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. Towards scalable translation validation of static analyzers. Technical Memorandum ROSAEC-2014-003, Research On Software Analysis for Error-free Computing Center, Seoul National University, November 2014. `http://ropas.snu.ac.kr/sparrowberry/`.
16. Jong-Gwon Kim, Woosuk Lee, Jaeseung Choi, Chung-Kil Hur, and Kwangkeun Yi. Shovel: A sat-based tool for information flow alarm classification. `http://sf.snu.ac.kr/shovel/`.
17. Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, July 2006.
18. Patrick Lam, Eric Bodden, Laurie Hendren, and Technische Universitt Darmstadt. The soot framework for java program analysis: a retrospective.
19. Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.
20. MITRE. *Common Vulnerabilities and Exposures (CVE) 2015-8106*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8106`.
21. MITRE. *Common Vulnerabilities and Exposures (CVE) 2015-8107*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8107`.
22. Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, January 2009.
23. George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.
24. George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.
25. Hakjoo Oh, Lucas Brutschy, and Kwangkeun Yi. Access analysis-based tight localization of abstract memories. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 356–370, Berlin, Heidelberg, 2011. Springer-Verlag.
26. Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. Global sparse analysis framework. *ACM Trans. Program. Lang. Syst.*, 36(3):8:1–8:44, September 2014.
27. Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for c-like languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 229–238, New York, NY, USA, 2012. ACM.

28. Hakjoo Oh and Kwangkeun Yi. Access-based localization with bypassing. In *Proceedings of the 9th Asian Conference on Programming Languages and Systems*, APLAS'11, pages 50–65, Berlin, Heidelberg, 2011. Springer-Verlag.
29. David Pichardie. Building certified static analysers by modular construction of well-founded lattices. *Electronic Notes in Theoretical Computer Science*, 212:225 – 239, 2008.
30. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28 – April 4, 1998 Proceedings*, pages 151–166, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.