Weakly Sensitive Analysis for Unbounded Iteration over JavaScript Objects

Yoonseok Ko[1], Xavier Rival[2], and Sukyoung Ryu[1]

[1] School of Computing, KAIST
[2] DIENS, École Normale Supérieure, CNRS, PSL Research University and INRIA

Abstract. JavaScript framework libraries like jQuery are widely used, but complicate program analyses. Indeed, they encode clean high-level constructions such as class inheritance via dynamic object copies and transformations that are harder to reason about. One common pattern used in them consists of loops that copy or transform part or all of the fields of an object. Such loops are challenging to analyze precisely, due to weak updates and as unrolling techniques do not always apply. In this paper, we observe that precise field correspondence relations are required for client analyses (e.g., for call-graph construction), and propose abstractions of objects and program executions that allow to reason separately about the effect of distinct iterations without resorting to full unrolling. We formalize and implement an analysis based on this technique. We assess the performance and precision on the computation of call-graph information on examples from jQuery tutorials.

1 Introduction

As JavaScript became popular, JavaScript libraries such as jQuery were also widely adopted. Among others, these libraries implement high-level programming constructions like module systems or class inheritance. While framework libraries are prevalent and contribute to making programs modular and reusable, they also make it more difficult to statically compute semantic properties of JavaScript programs such as type information and call graphs. This is due to the conversion of static high-level constructions into JavaScript dynamic constructions like objects [8]. Abundant works have been carried out so as to compute static information for type checking [14], optimization [11], program understanding [10], and security auditing [20]. However, they are not yet able to analyze JavaScript framework libraries precisely due to the highly dynamic features.

One of the problematic dynamic features is the read / write access to object fields the names of which are computed at runtime. In JavaScript, an object has a set of fields, each of which consists of a name and a value pair, and the field lookup and update operations take the value of an expression as an index that designates the field to read or write. JavaScript framework libraries exploit this facility to build an object from another object by copying all fields one by one (shallow copy) or by computing fields defined by the source object (e.g., to simulate accessor methods). To reason over such *field copy or transformation*

```
 1  function fix( e ) {
 2    i = copy.length;
 3    while ( i-- ) {
 4      p = copy[ i ];
 5      t = oE[ p ];
 6      e[ p ] = t;
 7    }  // oE.x ⋈ e.x, oE.y ⋈ e.y
 8  }
 9  var oE = {x: f1, y: f2}, o = {};
10  fix( o );
11  o.x();
```

Fig. 1: A simplified excerpt from jQuery 1.7.2 and an example use case

patterns (for short, FCT patterns), analysis tools need to resolve precisely the fields of objects, the relations among them, and how they encode the higher-level programming constructions. This is considerably harder than the resolution of classes in a direct Java-like implementation.

As an example, we consider the jQuery implementation of class-inheritance based on the field copies of the fields of an object, as shown in Figure 1. To achieve that, the function `fix` copies the fields of `oE` designated as the elements of an array `copy` into an array `e`. This copy of the fields of object `oE` takes place in the loop at lines 3–7. More generally, an FCT pattern boils down to an assignment of the form `o1[`f`(v)] = `g`(o2[v])`, where `v` is a variable, `o1` and `o2` are two objects, f is a function over field names and g is a function over values. We call the variable `v` the *key variable* of the FCT pattern. In Figure 1, the statements at lines 5 and 6 define an FCT pattern `e[p] = oE[p]` with the key variable `p`, where f and g both are the identity function.

When fields that get copied store closures, the static resolution of function calls requires to identify precise field relations. This is the case of the call at line 11, in the code of Figure 1. Therefore, the computation of a call-graph for this code requires a precise analysis of the effect of the FCT pattern at lines 5 and 6. Indeed, to determine the function called at line 11, we need to observe that the value of the field `x` in the object `e` is a copy of the value in the field `x` of the object `oE`. We note `oE.x` $\bowtie$ `e.x` for this *field correspondence relation*. A basic points-to analysis that applies no sensitivity technique on the loop (such as loop unrolling) will fail to establish this field correspondence relation, hence will not allow to determine to compute call-graph. Indeed, it would consider that at line 11, `o.x()` may call not only `f1` (that is actually called when executing the program), but also the spurious callee `f2`.

In this paper, we design a static analysis for the inference of precise field correspondence relations in programs that define FCT patterns. The inference of precise field correspondence relations is especially difficult when it takes place in loops (as in Figure 1, line 3), due to the possibly unbounded number of replicated fields. This complicates significantly the analysis of meta-programming framework libraries like jQuery. To achieve this, we observe that, while some

```
for (var v in o1) {
  o2[v] = o1[v];
}
```

(a)

```
var i = arr.length;
while ( i-- ) {
  v = arr[i];
  o2[v] = o1[v];
}
```

(b)

Fig. 2: Examples of loops with FCT patterns

loops can be analyzed using existing sensitivity techniques, others require a more sophisticated abstraction of the structure of objects, which can describe field correspondence relations over unbounded collections of fields. A first application of our analysis is the static computation of precise call-graph information, but it would also apply to other analyses, that also require tracking values propagated in FCT patterns. We make the following contributions:

– in Section 2, we categorize loops that perform field copies and transformations to two sorts, and characterize the techniques adapted for each of them;
– in Section 3, we construct a new analysis to infer precise field correspondence relations for programs that perform field copies and transformations;
– in Section 4, we evaluate our analysis on micro-benchmarks and real code.

2 Background

In this section, we study the categories of loops that contain FCT patterns, and discuss to what extent existing techniques can cope with them. Based on this discussion, we emphasize the need for different semantic abstraction techniques, to deal with cases that cannot be handled well.

Categories of loops with FCT patterns. A very common static analysis technique for loops proceeds by full or partial unrolling. Thus, we seek for categories of loops containing FCT patterns into two groups, depending on whether loop unrolling will allow to resolve impacted fields precisely. For instance, let us consider a loop the body of which contains an FCT pattern `o1[`f`(v)] = `g`(o2[v])`. Then, field correspondence relations are parameterized by the value of the key variable `v`. Thus, to infer such relations precisely, we need a precise characterization of the range of values of `v`. When `v` is the loop index, and the number of iterations is fixed to a known number N, we can simply unroll the loop fully N times.

JavaScript features both `for-in` and `while` loops, and we need to consider both kinds. Figure 2(a) displays a full FCT pattern based on a `for-in` loop: all fields of `o1` are copied into `o2`. Figure 2(b) shows a partial FCT pattern that is based on a `while` loop: the loop selects only fields that belong to `arr`. In both cases, to compute a precise field correspondence relation, we need to identify precisely the impacted fields and the relation between source and target objects.

When all iterations can be fully enumerated (e.g., by loop unrolling), it is possible to achieve this. In the case of the program in Figure 2(a), this is the case when the names and values of the fields of the object `o1` are known exactly from the pre-condition. We call such an instance a "concrete" `for-in` loop (which we abbreviate as CF). By contrast, when the set of fields is not known statically in the pre-condition, simple loop unrolling will not be sufficient to resolve fields precisely. We call such a case "abstract" `for-in` loop (or, for short, AF). The same distinction can be made for the `while` loop of Figure 2(b): when the pre-condition guarantees a fixed and known set of elements for the array `arr`, unrolling can make the analysis fully precise, and we call this configuration a "concrete" general loop (CG); otherwise, we call it "abstract" (AG).

We now briefly discuss the consequences of the imprecision that can follow from AF and AG loops, where no loop unrolling technique will prevent the weak updates in the analysis of FCT patterns in the loop body. Essentially, relations between field names and field values will be lost, as values of the key variable `v` range over a set that is statically undetermined. To illustrate this, we consider the case of Figure 2(b), and assume that the object `o1` has two fields named `x` and `y`, respectively storing closures `fx` and `fy` as values. Due to weak updates, the analysis of the partial copy loop will produce spurious field correspondence relations such as `(o1.x` $\bowtie$ `o2.y)`. As a consequence, the control flow analysis of a subsequent statement `o2.y()` would produce a spurious call edge to `fx`. Thus, call-graph information would be much less precise.

Analysis applied of loops with FCT patterns. When a loop that contains an FCT pattern can be categorized as CG or CF, unrolling is possible, and will allow strong updates. Thus, the above imprecision is eliminated. This technique is used in the state-of-the-art JavaScript analyzers such as TAJS [1] and $\mathsf{SAFE}_{\mathsf{LSA}}$ [16]. Since many of the loops in simple JavaScript programs are categorized as CG or CF, the dynamic unrolling approach has been successful to infer precise field correspondence relations so far. On the other hand, these sensitivity and unrolling based techniques cannot cope precisely with AF and AG loops. We note that the categorization of a loop as CF or AF (resp., as CG or AG) depends on the pre-condition of the loop, thus the computation of a more precise pre-condition may allow to reduce the analysis of an "abstract" loop to that of a "concrete" one. However, this is not always feasible. In particular, when the set of inputs of a program is infinite, and causes the pre-condition to include states with objects that may have an unbounded number of fields, this is not doable.

In the rest of the paper, we devise a new abstraction that can handle more precisely FCT patterns to compute more accurate field correspondence relations. The key idea is to let the analysis of one iteration of a loop with an FCT pattern `o1[`f`(v)] = `g`(o2[v])` describe a set of concrete iterations with the same value of the key variable. Since iteration orders do not matter for inferring the relations, one can abstract a set of non-consecutive iterations to an abstract iteration.

3 Composite Abstraction

We provide a high-level description of a composite abstraction to compute a precise over-approximation of field correspondence relations over FCT patterns.

3.1 Overview

We overview our approach with the analysis of an AG loop following the terminology of Section 2. We consider the program of Figure 2(b) assuming that the array `arr` is unknown and that `o1` has two fields with names `x` and `y`. This program carries out a partial copy of fields of `o1` into object `o2` depending on the contents of the array `arr`. Under this pre-condition, the loop is indeed AG, since the fields to be copied are not known statically. The variable `v` is key here, as it stores the value of the indices of the fields to consider.

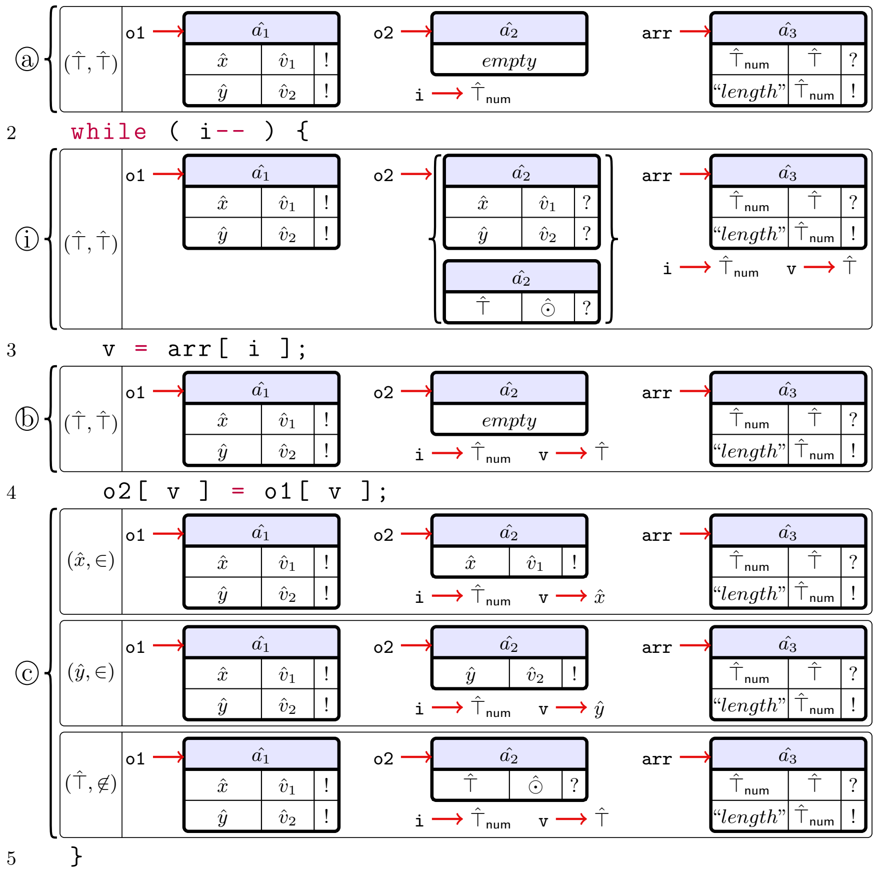
Even though `arr` is unknown, the program may copy only the fields in `o1`: the field `x` is either absent from `o1` or copied into `o2`, and the same for the field `y` (for short, we call the field designated by the name `v` "the field `v`" and use this terminology throughout the paper). Therefore, the field correspondence relation at the end of the execution of the loop may be any subset of $\{(\texttt{o1.x} \bowtie \texttt{o2.x}),\ (\texttt{o1.y} \bowtie \texttt{o2.y})\}$.

Abstraction. We first study the predicates that the analysis needs to manipulate, and the abstraction that they define. In general, we let *abstract addresses* used in the analysis be marked with a hat symbol, as in $\hat{a}$. Since the set of addresses that occur in a program execution is not known statically and is unbounded, the analysis needs to manipulate *summary* abstract addresses that may denote several concrete addresses. The analysis of a write into a summary address needs to account for the effect of a modification to any address it describes, thus it will result in a *weak update* [3]. On the other hand, a write into a *non-summary* address can be handled with a more precise *strong update*.

We need to tie to abstract addresses abstractions of sets of concrete objects that may not have exactly the same set of fields. Therefore, we let abstract objects collect *abstract fields* that may occur in the concrete objects they describe with an annotation that says whether the field must appear: "!" (resp., "?") designates a field that *must* (resp., *may*) exist. We also let abstract objects map fields that *must not* exist to `undefined` (or for short, $\hat{\odot}$).

In Figure 3, the abstract states ⓐ, ⓑ, and ⓒ show invariants at each program point in the first iteration over the loop. The state ⓐ shows an initial abstract state, where `o1`, `o2`, and `arr` point to abstract objects at addresses $\hat{a}_1$, $\hat{a}_2$, and $\hat{a}_3$, respectively, which are non-summary abstract addresses. The object `o1` has two abstract fields $\hat{x}$ and $\hat{y}$, and their values are $\hat{v}_1$ and $\hat{v}_2$. The field names $\hat{x}$ and $\hat{y}$ are non-summary abstract values which respectively denote the strings "x" and "y", and the values $\hat{v}_1$ and $\hat{v}_2$ are the values of the variables `v1` and `v2`. The fields of `o1` are annotated with !, as we know that the fields of `o1` is exactly $\{\texttt{x}, \texttt{y}\}$. The array `arr` has a definitely existing field `"length"` and its value is an unknown number. The other abstract field ($\hat{\top}_{\mathsf{num}}$, $\hat{\top}$, ?) expresses that the array has 0 or many elements, with unknown value. In the initial state, the object `o2` is empty, and the value of `i` is an unknown number $\hat{\top}_{\mathsf{num}}$.

The state ⓘ designates both the loop invariant and the post-condition of the program. We observe that the fields of `o2` are annotated with a question mark, which means they may or may not appear. We note that it captures the



```
while ( i-- ) {
  v = arr[ i ];
  o2[ v ] = o1[ v ];
}
```

Fig. 3: Invariants for the program of Figure 2(b): loop invariant at ⓘ and abstract states in the first iteration for all the other points.

intended field correspondence relations (`o1.x` $\bowtie$ `o2.x`) and (`o1.y` $\bowtie$ `o2.y`): if `o2` has a field `x`, the value of that field is the same as `o1.x` (and the same for `y`).

Analysis algorithm. We now present an analysis to compute invariants such as those shown in Figure 3. It proceeds by forward abstract interpretation [4], using the abstract states described above. It starts with the abstract state ⓑ that describes the initial states: `v` stores an unknown value $\hat{\top}$ and `i` stores an unknown number (as `arr` is unknown). It then progresses through the program forwards, accounts for an over-approximation of the effect of each statement, and computes loop invariants as the limit of sequences of abstract iterations over the effect of the loop body.

The first challenge is to analyze precisely the effect of one iteration of the loop body. For the field lookup operation `o1[v]` at line 4, we expect three possible results: $\hat{v}_1$, $\hat{v}_2$, and `undefined` because the object `o1` has two fields, $\hat{x}$ and $\hat{y}$, and the lookup operation will return `undefined` when it attempts to read an undefined field. Since the value of `v` subsumes the inputs of those three results, and in each case, the loop body has a different effect, the analysis needs to split the states into three sets of states characterized by the result of the lookup operation. In abstract interpretation terms, this means the analysis should perform some *trace partitioning*, so as to reason over the loop body using disjunctive properties. More precisely, the analysis should consider three cases:

1. If `v` $\in$ `o1` and `v` is $\hat{x}$, since the value of `v` designates an abstract field of `o1` that corresponds to exactly *one* concrete cell denoted by a non-summary address, this abstract field is copied to `o2` as a strong update. Therefore, at the end of the loop body, `o2` contains an abstract field $(\hat{x}, \hat{v}_1, !)$, as described in the first component $\{(\hat{x}, \in)\}$ of the abstract states ⓒ.
2. The case where `v` $\in$ `o1` and `v` is $\hat{y}$ is similar (the second one at ⓒ).
3. If `v` $\notin$ `o1`, the result of `o1[v]` is `undefined` since the value of `v` does not designate a field in `o1`. Thus, after line 4, `o2` may contain abstract fields designated by the value of `v` whose values are `undefined`. The last state $(\hat{\top}, \notin)$ of ⓒ corresponds to this case.

This partitioning operation turns one abstract state into a disjunction of several abstract states that cover the initial abstract state. In our implementation, this partitioning is carried out into two stages: first, the analysis discriminates executions depending on whether `v` is the name of a field of `o1`; second, when `v` is the name of a field of `o1`, the analysis produces a disjunction over the abstract fields of `o1`.

The second challenge is to compute precise approximations for unions of sets of states, as *abstract joins* (merging of partitions and control flow paths) and *widening* (union of abstract iterates to let the analysis of loops converge).

First, we observe that joining partitioned abstract states early would cause a precision loss, so that the analysis needs to delay their join. Indeed, let us assume that the analysis joins the results that correspond to the three sets of states at the end of the loop body at line 5. Then, the resulting abstract object would become the one in Figure 4, and the third field $(\hat{\top}, \{\hat{v}_1, \hat{v}_2, \hat{\odot}\}, ?)$ means that the value of any field in the object is either $\hat{v}_1$, $\hat{v}_2$, or $\hat{\odot}$; this abstraction fails to capture the precise field correspondence relation. Thus, instead of joining all the analysis results at the end of the loop body, our analysis maintains two abstract objects for the object pointed by `o2` as described in ⓘ.

However, in order to avoid overly large sets of disjuncts caused by delayed joins, the analysis also requires an efficient join algorithm, able to merge abstract states that are similar enough so that the imprecision shown in Figure 4 can be avoided. To do this, we rely on a join condition that states when abstract objects are "close enough" to be joined without a significant precision loss. Essentially, our analysis does not join objects with possibly overlapping abstract fields, the values of which are not similar. For example, at ⓒ, the abstract objects `o2`

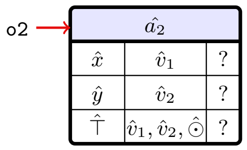


Fig. 4: The join of all the results at the end of the loop body in Figure 3.

$(\hat{x}, \in)$ and $(\hat{y}, \in)$ satisfy the join condition and are joined. On the contrary, at the same point, the abstract object $(\hat{\top}, \notin)$ is not joined with any other.

To ensure the convergence of abstract iterates, a widening operation over abstract objects is necessary. The widening algorithm is based on a similar principle as the join algorithm, but instead of merging two abstract objects, the analysis applies widening to them. Since the widening algorithm also relies on a join condition, it also does not lose the field correspondence relation. For example, at the loop head after the second iteration, the analysis applies widening to the state from the first iteration ⓐ and the state from the second iteration ⓘ. The widening finds two matchings for `o2` in ⓐ and ⓘ: the empty object in ⓐ with each of the objects in ⓘ. Because each field does not need widening, the result of widening for those two matchings are the same as the one in `o2` in ⓘ.

In the rest of the section, we formalize our composite abstraction to reason over FCT patterns and to infer precise field correspondence relations. It consists of two layers. The partitioning abstraction (Section 3.2) allows case splits, and the object abstraction (Section 3.3) describes accurate properties over fields, so as to disambiguate the field correspondence relations even after abstract joins.

3.2 Trace Partitioning Abstraction

A partitioning abstraction splits a set of traces based on a specific trace abstraction so as to reason more accurately on smaller sets of executions. We first set up a general partitioning framework, then define two instances with two trace abstractions that provide the two stages of partitioning we used in Section 3.1.

First, we fix a few notations. A concrete *program state* $s \in \mathbb{S}$ is a pair made of a *control state* (program counter at a given time) $c \in \mathbb{C}$ and a *memory state* $m \in \mathbb{M}$, thus $\mathbb{S} = \mathbb{C} \times \mathbb{M}$. We write $c \approx c'$ to denote that the syntactic program points of two control states c and c' are the same. A trace σ is a finite sequence $\langle s_0, ..., s_n \rangle$ where $s_0, ..., s_n \in \mathbb{S}$. We write $\mathbb{S}^\star$ for the set of traces, and let our concrete domain be $(\mathbb{D}, \sqsubseteq) = (\mathcal{P}(\mathbb{S}^\star), \subseteq)$. Also, if σ is a trace, we write $\sigma_{\rfloor M}$ for the memory state of the last state in σ. A trace partitioning abstraction will be an instance of the following definition:

Definition 1 (Partitioning Abstraction). *We let* $\mathbb{D}^\sharp_T$ *and* $\gamma_T : \mathbb{D}^\sharp_T \to \mathbb{D}$ *define a trace abstraction that is covering (i.e.,* $\mathbb{D} = \cup\{\gamma_T(t^\sharp) \mid t^\sharp \in \mathbb{D}^\sharp_T\}$*) and we assume that* $\mathbb{D}^\sharp_M$ *and* $\gamma_M : \mathbb{D}^\sharp_M \to \mathcal{P}(\mathbb{M})$ *defines a store abstraction. Then, the* partitioning abstraction *parameterized by these two abstractions is defined* *by the domain $\mathbb{D}^\sharp = \mathbb{D}^\sharp_T \to \mathbb{D}^\sharp_M$ and the concretization function γ defined by:*

$$\begin{aligned}\gamma : \mathbb{D}^\sharp &\longrightarrow \mathbb{D}\\ \hat{X} &\longmapsto \{\sigma \in \mathbb{S}^\star \mid \forall t^\sharp \in \mathbb{D}^\sharp_T,\ \sigma \in \gamma_T(t^\sharp) \Rightarrow \sigma_{\rfloor M} \in \gamma_M(\hat{X}(t^\sharp))\}.\end{aligned}$$

Trace abstraction by field existence. The first instance of partitioning abstraction that we need abstracts traces depending on the existence or absence of a field in an object. It is based on the following trace abstraction:

Definition 2 (Abstract Trace by Field Existence). *The trace abstraction $\mathbb{D}^\sharp_{\mathsf{fe}}$ is defined by the following abstract elements and abstraction relations:*

– *a quadruple $(c, \mathtt{x}, \mathtt{y}, p)$, where $c \in \mathbb{C}$, $\mathtt{x}$ is a variable, $\mathtt{y}$ is a variable (pointing to an object) and $p \in \{\in, \notin\}$, denotes all the traces that went through the control state c with a memory state such that the membership of the field the name of which is the value of $\mathtt{x}$ in the object $\mathtt{y}$ is characterized by p;*

– *$\hat{\top}$ abstracts any trace.*

For example, an abstract trace $(c, \mathtt{x}, \mathtt{y}, \in)$ denotes a set of traces such that an object $\mathtt{y}$ has a field designated by the value of the variable $\mathtt{x}$ at a given control state c. The first two states $(\hat{x}, \in)$ and $(\hat{y}, \in)$ of ⓒ in Figure 3 are the states partitioned by the abstract trace $(c, \mathtt{v}, \mathtt{o1}, \in)$.

Trace abstraction by variable value. The second instance of partitioning abstraction required by our analysis discriminates traces based on the value of a variable at a specific control point, and is parameterized by a value abstraction $\mathbb{V}^\sharp_{\mathsf{val}}$:

Definition 3 (Abstract Trace by Variable Value). *The trace abstraction $\mathbb{D}^\sharp_{\mathsf{vv}}$ is defined by the following abstract element and abstraction relations:*

– *a triple $(c, \mathtt{x}, \hat{v})$, where $c \in \mathbb{C}$, $\mathtt{x}$ is a program variable, and $\hat{v}$ is an abstract value $\hat{v} \in \mathbb{V}^\sharp_{\mathsf{val}}$, describes the traces that visited c with a memory state such that the value of $\mathtt{x}$ can be described by $\hat{v}$;*

– *$\hat{\top}$ abstracts any trace.*

In Figure 3, the abstract state $(\hat{x}, \in)$ at ⓒ denotes the partitioned states by the abstract trace $(c, \mathtt{v}, \hat{x})$.

Combination of trace abstractions. Trace abstractions can be combined into a reduced product abstraction, which defines the two-stage partitioning abstraction discussed in Section 3.1:

Definition 4 (Combination of Abstract Traces). *An abstract trace characterized by a combination of two abstract traces is a reduced product [5] of their trace abstractions.*

Our analysis uses product $\mathbb{D}^\sharp_T \stackrel{\mathsf{def}}{=} \mathbb{D}^\sharp_{\mathsf{fe}} \times \mathbb{D}^\sharp_{\mathsf{vv}}$. For example, the three states at point ⓒ in Figure 3 correspond to the partitions defined by $((c, \mathtt{v}, \mathtt{o1}, \in), (c, \mathtt{v}, \hat{x}))$, $((c, \mathtt{v}, \mathtt{o1}, \in), (c, \mathtt{v}, \hat{y}))$, and $((c, \mathtt{v}, \mathtt{o1}, \notin), (c, \mathtt{v}, \hat{\top}))$.

$$\begin{array}{rrcl}
\gamma_{\mathbb{O}^\sharp}: & \mathbb{O}^\sharp & \longrightarrow & \mathcal{P}(\mathbb{O}) \\
& l & \longmapsto & \{o \in \mathbb{O} \mid o \in \gamma_l(l) \wedge \mathsf{Dom}(\mathsf{o}) \subseteq \mathsf{Dom}^\sharp_\mathsf{l}(\mathsf{l})\} \\[1ex]
\gamma_l: & p & \longmapsto & \gamma_p(p) \\
& l_0 \wedge l_1 & \longmapsto & \gamma_l(l_0) \cap \gamma_l(l_1) \\
& l_0 \vee l_1 & \longmapsto & \gamma_l(l_0) \cup \gamma_l(l_1) \\[1ex]
\gamma_p: & \epsilon & \longmapsto & \mathbb{O} \\
& (\hat{s} \mapsto !, \hat{v}) & \longmapsto & \{o \in \mathbb{O} \mid \forall s \in \gamma_{\text{str}}(\hat{s}),\ o(s) \in \gamma_{\text{val}}(\hat{v})\} \\
& (\hat{s} \mapsto ?, \hat{v}) & \longmapsto & \{o \in \mathbb{O} \mid \forall s \in \gamma_{\text{str}}(\hat{s}),\ s \in \mathsf{Dom}(o) \Rightarrow o(s) \in \gamma_{\text{val}}(\hat{v})\}
\end{array}$$

$$\begin{array}{rrcl}
\mathsf{Dom}^\sharp_l: & p & \longmapsto & \mathsf{Dom}^\sharp_p(p) \\
& l_0 \wedge l_1 & \longmapsto & \mathsf{Dom}^\sharp_l(l_0) \cup \mathsf{Dom}^\sharp_l(l_1) \\
& l_0 \vee l_1 & \longmapsto & \mathsf{Dom}^\sharp_l(l_0) \cup \mathsf{Dom}^\sharp_l(l_1)
\end{array}
\qquad
\begin{array}{rrcl}
\mathsf{Dom}^\sharp_p: & \epsilon & \longmapsto & \emptyset \\
& (\hat{s} \mapsto E, \hat{v}) & \longmapsto & \gamma_{\text{str}}(\hat{s})
\end{array}$$

Fig. 5: Concretization of abstract objects

3.3 Object Abstraction

The object abstraction forms the second layer of our composite abstraction, and describes the fields of JavaScript objects. A JavaScript object consists of a (non-fixed) set of mutable fields, and the main operations on the objects are field lookup, field update, and field deletion. A concrete object is a map $o \in \mathbb{O} = \mathbb{V}_{\text{str}} \rightarrow_{\mathsf{fin}} \mathbb{V}_{\text{val}}$, where $\mathbb{V}_{\text{str}}$ stands for the set of string values and $\mathbb{V}_{\text{val}}$ stands for the set of values. We write $\mathsf{Dom}(\mathsf{o})$ for the set of fields of an object o.

To accurately handle absent fields, our object abstraction can track cases where a field is definitely absent. Similarly, to handle lookup of definitely existing fields precisely, the object abstraction annotates such fields with the predicate "!". Fields that may or may not exist are annotated with the predicate "?": the lookup of such fields may return a value or the value `undefined`. In the following, we assume an abstraction for values defined by the domain $\mathbb{V}^\sharp_{\text{val}}$ and the concretization function $\gamma_{\text{val}} : \mathbb{V}^\sharp_{\text{val}} \rightarrow \mathcal{P}(\mathbb{V}_{\text{val}})$, and an abstraction for string values defined by the finite height domain $\mathbb{V}^\sharp_{\text{str}}$ and the concretization function $\gamma_{\text{str}} : \mathbb{V}^\sharp_{\text{str}} \rightarrow \mathcal{P}(\mathbb{V}_{\text{str}})$.

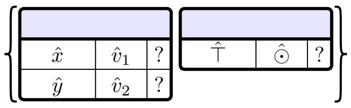
Definition 5 (Abstract Object). *An* abstract object $\hat{o} \in \mathbb{O}^\sharp$ *is a logical formula described by the following grammar:*

$$\begin{array}{lll}
\hat{o} ::= l & l ::= p \mid l \wedge l \mid l \vee l & \hat{s} \in \mathbb{V}^\sharp_{\text{str}} \\
p ::= \epsilon \mid (\hat{s} \mapsto E, \hat{v}) & E ::= \,! \mid ? & \hat{v} \in \mathbb{V}^\sharp_{\text{val}}
\end{array}$$

The concretization of $\gamma_{\mathbb{O}^\sharp}(\hat{o})$ of an abstract object $\hat{o}$ is defined in Figure 5.

An abstract field p is either ϵ, which represents an object with unknown fields, or $(\hat{s} \mapsto E, \hat{v})$, which represents any object such that the membership of fields the names of which are represented by $\hat{s}$ is described by E and the values of the corresponding fields (if any) by $\hat{v}$. Note that E may be ! (must exist) or ? (may exist). A logical formula l describes a set of objects, either as an abstract field p, or as a union or intersection of sets of objects. A conjunction of abstract fields represents a set of concrete open objects. A disjunction $l_0 \vee l_1$ represents two abstract objects l_0 and l_1 as for the object `o2` in the abstract state at point ⓘ in Figure 3. Because the objects represented by an abstract field p may have an unbounded number of fields, the concretization of abstract objects $\gamma_{\mathbb{O}^\sharp}$ bounds the set of fields with $\mathsf{Dom}(o) \subseteq \mathsf{Dom}^\sharp_l(l)$. The auxiliary function $\mathsf{Dom}^\sharp_l(l)$ represents a set of fields that are defined in a given logical formula l, which denotes that a field is definitely absent if the field is not defined in l. For example, the concretization of an abstract field $\gamma_p(\hat{x} \mapsto !, \hat{v})$ represents a set of objects, where the values of the fields designated by the abstract string $\hat{x}$ are represented by the abstract value $\hat{v}$ and the other fields are unknown, whereas the abstract object concretization $\gamma_{\mathbb{O}^\sharp}(\hat{x} \mapsto !, \hat{v})$ represents a set of objects all fields of which are abstracted by $\hat{x}$ (i.e., any field not described by $\hat{v}$ is absent). The conjunction and disjunction are conventional.

Then, the object box notation used in Figure 3 is interpreted as follows:



is equivalent to

$$((\hat{x} \mapsto ?, \hat{v}_1) \wedge (\hat{y} \mapsto ?, \hat{v}_2)) \vee (\hat{\top} \mapsto ?, \hat{\odot}).$$

3.4 Analysis Algorithms

We now study the analysis algorithms to infer invariants such as those shown in Figure 3. These rely on the trace partitioning abstraction defined in Section 3.2 and on the object abstraction defined in Section 3.3. We focus on the introduction of partitions and on the join algorithm to achieve precise analysis of the FCT patterns. We provide a formal description for a language and an algorithm in Appendices **??** and **??**.

Partitioning algorithm. As we have shown in Section 3.1, the analysis needs to create partitions dynamically. We first discuss the analysis of loops classified as AF, and consider the case of AG loops afterwards.

For-in loops. We desugar the `for-in` loop in Figure 2(a) with high-level `iterInit`, `iterHasNext`, `iterNext`, and `merge` instructions as shown in Figure 6(a). The semantics of the `for-in` statement is as follows: it iterates over the fields of a given object `o1`, and lets the index variable `v` range over their names. To mimic this behavior, the instruction `v = iterInit(o1)` generates the list of the fields of the object `o1`; `iterHasNext(t)` checks whether there exists a field that has not been visited yet, and `iterNext(o1, t)` returns the next field to visit. Since the order a `for-in` loop uses to visit fields is undefined in the language semantics [9], we simply assume a non-deterministic order of fields. We add the `merge(v)` pseudo-instruction at the end of each loop to denote the end of the FCT pattern with the key variable `v`. In order to reason over distinct fields separately, the analysis performs dynamic partitioning during the analysis of the `v = iterNext(o1, t)` instruction, and generates partitions for each possible field

```
var v, t = iterInit(o1);
while (iterHasNext(t)) {
  v = iterNext(o1, t);
  o2[v] = o1[v];
  merge(v);
}
```

(a)

```
var i = arr.length;
while ( i-- ) {
   v = arr[i];
   o2[v] = o1[v];
   merge(v);
}
```

(b)

Fig. 6: Desugared version of `for-in` and `while` loops shown in Figure 2.

name. Because the analysis needs to analyze field names in each partition precisely so that it can analyze the field lookup instruction at line 4 precisely, it collects a set of field names in the given object `o1`, and generates partitions for each field name in the set. In Figure 7(a), we illustrate such a case. At the end of the loop body, the partitions are merged by the `merge` instruction as shown in Figure 7(c).

General loops. We first perform a simple syntactic pre-analysis, which finds variables that are used as index variables in field lookup and update instructions in the loop body, in order to identify key variables of the FCT pattern. The pre-analysis adds the `merge(x)` instruction at the end of the loop when it identifies `x` as the key variable of an FCT pattern. We consider the `while` loop in Figure 2(b), and we show the result of the pre-analysis in Figure 6(b). In this instance, `v` is identified as a key variable as it appears for both a field lookup and a field update in the loop body. On the contrary, the variable `i` is not a key variable because it is only used for field lookup.

Based on the result of this pre-analysis, the analysis creates partitions when it encounters a field lookup indexed by a key variable. It essentially creates one partition per possible (defined or undefined) abstract value produced as the result of the lookup operation. For example, Figure 7(b) shows that for such cases when the field lookup may return either the field $\{(\hat{q}, \hat{v}_1)\}$ or the field $\{(\hat{w}, \hat{v}_2)\}$, or the `undefined` value $\{\hat{\odot}\}$ (if the field is absent in the object). The variable x_1 is temporarily created to hold the result of the lookup. Each generated partition corresponds to a field correspondence relation. Similarly for the `for-in` loop case, the partitions are merged at the end of the loop body.

Soundness of partitioning algorithm. The partitioning algorithm is sound since (1) the generated partitions for the `iterNext` instruction subsumes all possible iterations in a given `for-in` loop, and (2) the abstract lookup operation returns a disjunction of cases that subsume the result of the concrete lookup operation.

Join and widening algorithms. We now discuss the join and widening that allow to merge abstract partitions at `merge` pseudo instructions and to analyze loops. For instance, in Figure 3, the analysis computes the object `o2` at point ⓘ by joining the three objects `o2` of ⓒ.

To simplify the algorithms, we make sure that each abstract object is in "disjunctive normal form", which means that its logical formula is a disjunction

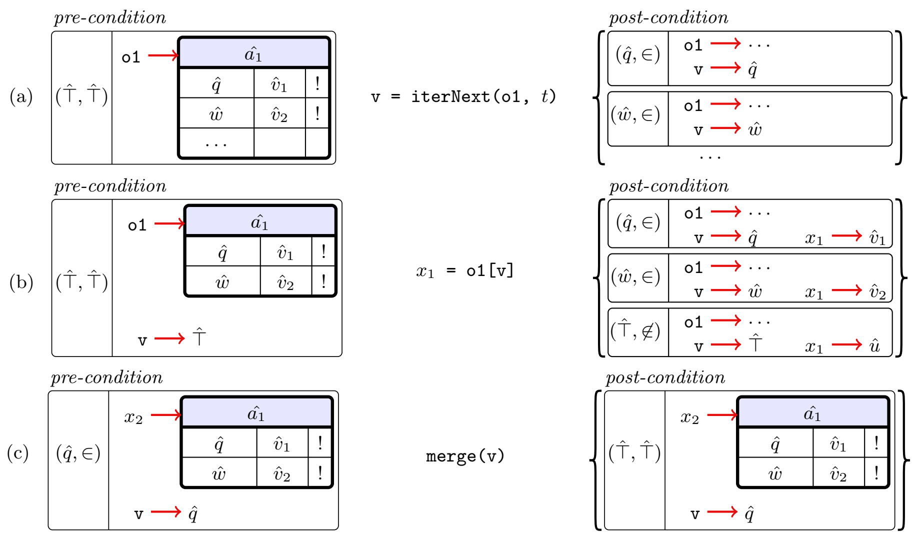

pre-condition
post-condition
(a)
$(\hat{\top}, \hat{\top})$
o1
$\hat{a}_1$
$\hat{q}$ $\hat{v}_1$!
$\hat{w}$ $\hat{v}_2$!
v = iterNext(o1, t)
$(\hat{q}, \in)$ o1 → ... v → $\hat{q}$
$(\hat{w}, \in)$ o1 → ... v → $\hat{w}$
(b)
x_1 = o1[v]
v → $\hat{\top}$
$(\hat{q}, \in)$ o1 → ... v → $\hat{q}$ x_1 → $\hat{v}_1$
$(\hat{w}, \in)$ o1 → ... v → $\hat{w}$ x_1 → $\hat{v}_2$
$(\hat{\top}, \notin)$ o1 → ... v → $\hat{\top}$ x_1 → $\hat{u}$
(c)
$(\hat{q}, \in)$
x_2
v → $\hat{q}$
merge(v)
$(\hat{\top}, \hat{\top})$

Fig. 7: Example cases for (a) the `iterNext` instruction, (b) the field lookup instruction with the key variable `v`, and (c) the `merge` instruction by the key variable `v` with the first post-state of both of (a) and (b).

of "conjunctive abstract objects" that consist of a conjunction of abstract fields or of the empty abstract field ϵ. The join algorithm takes two abstract objects in this form and computes an over-approximation of all the concrete objects they represent. For two such abstract objects `o0` and `o1`, the join algorithm searches for a matching of the conjunctive abstract objects in `o0` and `o1`, that can be merged without too significant a precision loss, as characterized by a *join condition* that we discuss below. When `c0` and `c1` satisfy this condition and can be joined, they are replaced with a new conjunctive abstract object, where each abstract field approximates abstract fields of `c0` and `c1`.

We now make this join condition used to decide which conjunctive objects to join. Let us consider the join of a pair of abstract fields. When the name of one of these fields subsumes the name of the other but is not equal to it, their names and values are merged and the result fails to capture an optimal field correspondence relation. For example, the result of joining two abstract fields $(\hat{x}, !, \hat{v}_1)$ and $(\hat{\top}, ?, \hat{v}_2)$ is $(\hat{\top}, ?, \{\hat{v}_1, \hat{v}_2\})$ because the name $\hat{\top}$ subsumes the name of the other field $\hat{x}$. Thus, we avoid joining abstract objects containing such pairs of abstract fields as characterized by the definition below:

Definition 6 (Join Condition). *Two sets of abstract fields $\hat{P}_1$ and $\hat{P}_2$ are* correlated *if and only if* $\forall \hat{s}_1 \in N(\hat{P}_1)$, $\forall \hat{s}_2 \in N(\hat{P}_2)$, $\gamma_{\text{str}}(\hat{s}_1) \cap \gamma_{\text{str}}(\hat{s}_2) \neq \emptyset$ *where $N(\hat{P})$ (resp., $V(\hat{P})$) denotes the names (resp., values) of the fields in $\hat{P}$.*

Two abstract objects $\hat{o}_1$ and $\hat{o}_2$ satisfy the join condition *if and only if, for all the correlated abstract fields $\hat{P}_1$ and $\hat{P}_2$ in $\hat{o}_1$ and $\hat{o}_2$, one of the following conditions holds:*
- $\hat{N}_1 = \hat{N}_2$
- $\hat{N}_1 \subseteq \hat{N}_2 \wedge \hat{V}_1 \sqsubseteq \hat{V}_2$*, or, symmetrically* $\hat{N}_2 \subseteq \hat{N}_1 \wedge \hat{V}_2 \sqsubseteq \hat{V}_1$
- $\hat{N}_1 \cap \hat{N}_2 \neq \emptyset \wedge \hat{V}_1 = \hat{V}_2$

where $\hat{N}_1 = \{\gamma_{\mathrm{str}}(\hat{s}) \mid \hat{s} \in N(\hat{P}_1)\}$, $\hat{N}_2 = \{\gamma_{\mathrm{str}}(\hat{s}) \mid \hat{s} \in N(\hat{P}_2)\}$, $\hat{V}_1 = \cup\, V(\hat{P}_1)$, *and* $\hat{V}_2 = \cup\, V(\hat{P}_2)$.

For example, the object o2 in the third disjunct at ⓒ does not satisfy the join condition with the object o2 in either of the other two disjuncts at that point. On the other hand, the abstract objects corresponding to o2 in the first two disjuncts at ⓒ satisfy this condition, and will be joined together.

Join soundness. The abstract join algorithm is sound. It returns a conservative over-approximation of its inputs when the join condition holds; otherwise, it returns their disjunction, the result of which also over-approximates them.

Widening and termination. The analysis of a loop proceeds by computation of a sequence of abstract iterates. To ensure its convergence, we use a widening operator [4], that is, an operator that computes an over-approximation of union and ensures termination. Since the height of the baseline string abstraction is finite, the second join condition ($\hat{N}_1 \subseteq \hat{N}_2$ or $\hat{N}_2 \subseteq \hat{N}_1$) can be satisfied for at most finitely many iterations. This entails that any sequence of abstract joins is ultimately stationary, which ensures the termination of the analysis of loops.

Analysis termination and soundness. The analysis terminates and computes a sound over-approximation for the set of concrete program behaviors. Thus, it computes a conservative approximation of the call-graph of an input program.

4 Evaluation

This section seeks for an experimental validation that the composite abstraction is necessary and useful for the computation of precise call-graph information.

Analysis implementation and experimentation settings. We implemented the analyzer CompAbs, based on our new composite abstraction. CompAbs was built on top of the open-source JavaScript analysis framework SAFE [15]. CompAbs is fully context sensitive except for recursive calls, which are unrolled at most j times, where j is a parameter of the analysis. Dynamic unrolling of loops is also bounded by a parameter k. Moreover, CompAbs abstracts strings based on creation-site, and on a reduced product of a string constant domain and a prefix string domain. We used different parameters for different experiments. We performed all the experiments on a Linux machine with 4.0GHz Intel Core i7-6700K CPU and 32GB Memory.

Analysis of micro benchmarks with different loop patterns. The goal of this experiment is to observe the current status of the state-of-the-art JavaScript static analyzers TAJS and $\mathsf{SAFE}_{\mathsf{LSA}}$ for each category of loops introduced in Section 2.

We ran TAJS, $\mathsf{SAFE}_{\mathsf{LSA}}$, and CompAbs on a number of small diagnostic benchmarks, each of which consists of a loop containing an FCT pattern. We manually wrote the test cases to assess whether an analysis can precisely infer a field correspondence relation or not for each category of loops. The benchmarks do not use DOM-related features. They are set up so that each loop is being classified in the same category in all of three analyzers TAJS, $\mathsf{SAFE}_{\mathsf{LSA}}$, and CompAbs. At the end of each loop, we attempt to verify the precise field correspondence relation by equality check between fields ($\texttt{o.x} \neq \texttt{o.y}$). Since we compare the analysis result on small benchmarks, CompAbs used $j = 0$ and $k = 0$ as parameters, which leads to merging all recursive calls and no dynamic unrolling.

The inset table summarizes the result of the micro benchmarks whether each analyzer can precisely infer the field correspondence relation or not. The first column shows the FCT pattern category of each single loop program. The remaining columns report whether each analysis successfully infers the precise field correspondence relation or not. TAJS and $\mathsf{SAFE}_{\mathsf{LSA}}$ fail to infer precise field correspondence relations in the case of "abstract" loops (AF or AG), which is consistent with our observation that such loops cannot be addressed by using only the unrolling techniques implemented in these tools.

Subject	TAJS	$\mathsf{SAFE}_{\mathsf{LSA}}$	CompAbs
CF	✓	✓	✓
CG	✓	✓	✓
AF	✗	✗	✓
AG	✗	✗	✓

This confirms that the existing analyzers fail to infer precise field correspondence relations for loops with an FCT pattern, which are classified as "abstract".

Analysis of jQuery tutorials. The goal of this experiment is to compare our approach with the existing analyzers using jQuery tutorial benchmarks. We measured the number of programs that have "abstract" loops to assess the necessity of our approach. We also measured the number of programs that can be analyzed in a given time limit to assess the scalability, and measured the number of spurious call edges in the result to assess the precision. To evaluate the analysis of FCT patterns in framework libraries, we use the jQuery benchmarks from the experiments of TAJS [1]. They used 71 programs chosen from the jQuery tutorial[3] that performs simple operations using jQuery 1.10.0. In order to minimize the effects of DOM modeling on the analysis results, our experiments use jQuery 1.4.4 that supports all the features required for the benchmarks while using fewer DOM-related features than jQuery 1.10.0. We faithfully implemented DOM models based on a Chrome browser to support the semantics of DOM-related code in our benchmarks. We do not abstract the DOM tree in the initial state, and abstract a set of dynamically generated DOM elements to an abstract DOM element depending on its allocation site. The programs are about 7430 lines of code including jQuery.

[3] `http://www.jquery-tutorial.net/`

Table 1: Scalability and precision for call graphs compared to existing analyzers.

	Success	SE	SC	Cov(%)
CompAbs	68 / 71	0.03	14.9	99.8
TAJS	2 / 71	6.00	132.8	77.3
$\mathsf{SAFE}_{\mathsf{LSA}}$	0 / 71	45.96	51.1	81.7

First, to investigate the necessity of our work, we measured how many programs have "abstract" loops during analysis. In order to compute a precise precondition to reduce the analysis of an "abstract" loop to that of a "concrete" one, CompAbs used $j = 2$ and $k = 30$ as parameters, which leads to unrolling recursive calls twice and bounding dynamic unrolling depth to 30 (when it can be applied). We chose the parameters by experiments. Among 71 benchmarks, we observed that 27 programs have at least one abstract loop. Thus, about 38% of the benchmarks need to handle abstract loops, which may require our technique.

Second, to compare the analysis scalability, we measured the number of programs the analysis of which terminates in less than 4 hours by CompAbs, TAJS, and $\mathsf{SAFE}_{\mathsf{LSA}}$. The second column in Table 1 summarizes the scalability measurements; out of 71 benchmarks, CompAbs finished analysis of 68 programs among these within 4 hours; TAJS finished analysis of 2 programs among these; and $\mathsf{SAFE}_{\mathsf{LSA}}$ did not finish analysis of any of these 71 programs. CompAbs finished analysis of each of 64 programs within 6 minutes, and it took 27 seconds on average (minimum, median, and maximum are 5, 9, and 315 seconds, respectively) for each program. Among the 7 other programs, CompAbs analyzed each of 4 programs within 4 hours, and analyzed each of the rest of 3 programs within 2.1 days. Our manual investigation revealed that two reasons caused the analysis performance overhead: (1) a large number of recursive call cycles, and (2) long call chains. The large number of recursive call cycles is a common problem in static analysis, and the techniques to mitigate the complexity are beyond the scope of this paper. Also, since CompAbs is fully context sensitive, longer call chains make the analysis take more time. On the other hand, TAJS finished analysis of 2 programs within 7 minutes. However TAJS did not complete the analysis of any of the 69 other programs within 4 hours. $\mathsf{SAFE}_{\mathsf{LSA}}$ did not finish analysis of any of these 71 programs within 4 hours.

At last, to assess the analysis precision, we measured the number of spurious call edges in analysis results. We identify spurious call edges in the analysis result of a program by using dynamic call graphs constructed by real execution of the instrumented program in a Chrome browser. Since the benchmarks are simple and we ran them several times trying to cover all the possible execution flows, we assume that the dynamic call graphs contain a complete set of call edges. Then, in the analysis results, if a call site has more than one call edge that do not appear in the dynamic call graphs, we consider them spurious call edges.

We found spurious call edges from eight subjects (39, 40, 44, 47, 50, 53, 69, and 71) out of 71. We manually investigated them, and observed that dynamic call graphs for five subjects (40, 44, 50, 53, 71) were incomplete because Chrome browser had an HTML parsing problem. We confirmed that the spurious call

Table 2: Scalability and precision for call graphs compared to dynamic call graphs.

	Time			SE	SC	Prec(%)	Cov(%)
	Avg	Med	Max				
CompAbs	5687 sec	10 sec	2.1 days	0.03	14.9	99.995	99.97
ACG	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	85.37	1061.5	65.148	84.39

edges from the five subjects disappeared after we revised the HTML documents to bypass the parsing problem. We found that each of the remaining 3 subjects had one callsite with spurious call edges, all of which are due to imprecise DOM modeling or infeasible execution flows. Because our DOM modeling does not consider the precise semantics of browser layout engines, the analysis cannot precisely analyze several DOM-related functions such as `querySelectorAll` or `getComputedStyle`. Imprecise analysis results of such function calls affect the analysis results of execution flows, which may cause spurious call edges.

The third to the fifth columns of Table 1 summarize the precision measurements of 71 benchmarks; SE denotes the number of spurious call edges on average, and SC denotes the number of spurious call sites on average; Cov denotes the coverage ratio of actual call sites. Since we measured the number of spurious call edges from an incomplete analysis result when an analyzer does not finish analyzing a program in 4 hours, the average coverage ratio is not 100%, and the actual number of spurious call edges may be larger than the one in the table. We use the revised HTML documents to bypass the browser parsing problem.

CompAbs outperforms TAJS and SAFE_{LSA} in analysis time and precision. TAJS and SAFE_{LSA} lose their precision and scalability when they analyze "abstract" loops. TAJS used a modified jQuery library by manually removing abstract field names that can cause "abstract" loops in their evaluation [1]. Since we do not use the modification in our benchmarks, the scalability of TAJS got worse than what they presented in their evaluation. Since the approach of SAFE_{LSA} to handle loops is similar to TAJS, SAFE_{LSA} has the same issue.

We also checked the soundness of CompAbs using dynamic call graphs. We confirmed that static call graphs of 70 out of 71 programs subsumed their corresponding dynamic call graphs. Because the subject (4) used `eval` that constructs code dynamically, static call graphs could not cover dynamically constructed call edges, for which CompAbs reports possible unsoundness warnings.

Comparison with an unsound approach. The goal of this experiment is to compare the results of a sound approach and an unsound one. We first evaluate CompAbs, which computes over-approximation of call graphs, and the practical unsound JavaScript call graph analyzer [10] by two metrics: the scalability and precision. We write ACG to denote the practical unsound call graph analyzer[4]. The experiment also uses the 71 jQuery benchmarks.

Table 2 summarizes the scalability and precision. The second to the forth columns show the average, the median, and the worst case analysis time, respectively. The remaining columns show the analysis precision; SE, SC, and Cov are

[4] `https://github.com/xiemaisi/acg.js`

as in Table 1, and Prec denotes the percentage of "true" call edges among all call edges, which is computed as $\frac{|D \cap S|}{|S|}$ where D is the set of call edges of a given call site in the dynamic call graph and S is the set of call edges determined by the analysis.

The analysis time of ACG is dramatically faster than CompAbs because it does not consider dynamic field lookup and update instructions, does not keep more than one abstract object, and does not reason about any non-functional values. On the other hand, since ACG does not compute over-approximation, the analysis cannot capture 15.61% of actual call-sites. In addition, while CompAbs identifies exactly what is missing in the branch coverage, ACG does not show what is missing, which is unsound. CompAbs computes more precise call graphs than that of ACG. Since CompAbs precisely tracks the program flows, it produces fewer spurious call sites and fewer spurious call edges.

Thus, ACG is more appropriate than CompAbs in supporting tools such as the "Jump to Declaration" feature in Integrated Development Environment(IDE). Since ACG does not require the full source code of a target program to analyze, it is applicable to incomplete source code which is under development. In the case of IDE usage, the increased scalability can compensate the low precision of the analysis and the unsoundness. While CompAbs supports call edges only in reachable code, ACG also supports call edges in dead code. In this case, the low precision of the number of spurious call sites is rather an advantage.

On the other hand, CompAbs is more appropriate than ACG in verifying semantic properties of JavaScript programs such as private information leakage. Since a verification tool requires zero false negatives, formalizing the analysis as abstract interpretation to compute over-approximations of program behaviors as in CompAbs is appropriate in this purpose. Also, the high-precision of CompAbs will be useful in reducing the number of false alarms in a verification tool. Verification tools do not need to provide instant feedback to developers, and we believe that the importance of verification compensates analysis time overhead.

5 Related Work

SAFE [15] is an analysis framework for JavaScript web applications, which supports DOM modeling [17] and web API misuse detection [2], among others. It is based on abstract interpretation, and supports various analysis sensitivities including k-CFA, parameter sensitivity, object sensitivity, and loop-sensitive analysis (LSA) [16]. While its aggressively unrolling LSA works well for programs using simple loops with bounded iterations, we showed that it does not scale for complex loops with unbounded numbers of iterations or imprecise pre-conditions.

TAJS [14,1] is an open-source static analyzer for JavaScript, which supports DOM modeling [13]. It uses a simple object abstraction, which can be represented by a limited form of conjunctions, without disjunctions. While the object abstraction is clearly designed and highly tuned for their own string abstraction, it does not support various string abstractions for a field name abstraction. In addition, as we described in Section 2, it may lose precision when it disambiguates field correspondence relations after joining the states from loop iterations.

HOO [6] computes a very different abstraction of JavaScript objects, which captures abstract forms of field relations. Indeed, HOO lets symbolic set variables denote unbounded size sets of fields of JavaScript objects to express logical facts on these sets, using specific set abstract domains [7]. For instance, it can express that two objects have exactly the same set of fields, or that the fields of a first object are included into the fields of a second object. Such program invariants are useful to reason over libraries that may be used in a general and unknown context, or in order to implement a modular verifier. By contrast, this approach is less adequate to compute precise information about the result of long and complex initialization routines like the codes our analysis targets.

WALA [12] supports static analysis of JavaScript programs. It performs a conventional propagation-based pointer analysis with techniques that can handle some of the FCT patterns [19,18].

ACG [10] is a field-based, intentionally unsound call graph analysis. The analysis ignores dynamic field lookup and update instructions, does not track non-functional values, and does not distinguish JavaScript objects since it abstracts all concrete objects into a single abstract object. Thus, ACG achieves a practical level of scalability and precision. Since ACG does not compute an over-approximation of program behaviors, the analysis is not appropriate to verify safety properties in a program. On the other hand, it is useful to support development tools.

6 Conclusion

We presented a composite abstraction that can capture precise field correspondence relations by local reasoning about loop iterations. The composite abstraction consists of two layers of abstractions: the partitioning abstraction that summarizes loop iterations in an iteration order independent way, and the object abstraction that accurately captures field correspondence relation of JavaScript objects. While most existing JavaScript analyses unroll loops aggressively, which does not scale for loops that cannot be unrolled, our analysis can infer field correspondence relations precisely even for loops which cannot be unrolled. The results of experiments based on a prototype implementation show that the analysis can avoid a large number of spurious callees caused by the precision loss of the field correspondence relation in jQuery examples.

Acknowledgment. The research leading to these results has received funding from the European Research Council under the EU FP 7, grant Agreement 278673, Project MemCAD.

Acknowledgment. The research leading to these results has received funding from Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grants NRF-2017R1A2B3012020 and NRF-

2014R1A2A2A01003235), Samsung Electronics, and the European Research Council under the EU FP 7, grant Agreement 278673, Project MemCAD.

References

1. Andreasen, E., Møller, A.: Determinacy in static analysis for jQuery. In: OOPLSA (2014)
2. Bae, S., Cho, H., Lim, I., Ryu, S.: $\mathsf{SAFE}_{\mathsf{WAPI}}$: Web API misuse detector for web applications. In: ESEC/FSE (2014)
3. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: SAS (2006)
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL (1979)
6. Cox, A., Chang, B.Y.E., Rival, X.: Automatic analysis of open objects in dynamic language programs. In: SAS (2014)
7. Cox, A., Chang, B.Y.E., Sankaranarayanan, S.: QUIC graphs: Relational invariant generation for containers. In: ECOOP (2013)
8. Eshkevari, L., Mazinanian, D., Rostami, S., Tsantalis, N.: JSDeodorant: Class-awareness for JavaScript programs. In: ICSE (2017)
9. European Association for Standardizing Information and Communication Systems (ECMA): ECMA-262: ECMAScript Language Specification. Edition 5.1 (2011)
10. Feldthaus, A., Schäfer, M., Sridharan, M., Dolby, J., Tip, F.: Efficient construction of approximate call graphs for JavaScript IDE services. In: ICSE (2013)
11. Hackett, B., Guo, S.y.: Fast and precise hybrid type inference for JavaScript. In: PLDI. New York, NY, USA (2012)
12. IBM Research: T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`
13. Jensen, S.H., Madsen, M., Møller, A.: Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In: ESEC/FSE (2011)
14. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: SAS (2009)
15. Lee, H., Won, S., Jin, J., Cho, J., Ryu, S.: SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In: FOOL (2012)
16. Park, C., Ryu, S.: Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In: ECOOP (2015)
17. Park, C., Won, S., Jin, J., Ryu, S.: Static analysis of JavaScript web applications in the wild via practical DOM modeling. In: ASE (2015)
18. Schäfer, M., Sridharan, M., Dolby, J., Tip, F.: Dynamic determinacy analysis. In: PLDI (2013)
19. Sridharan, M., Dolby, J., Chandra, S., Schäfer, M., Tip, F.: Correlation tracking for points-to analysis of JavaScript. In: ECOOP (2012)
20. Wei, S., Ryder, B.G.: Practical blended taint analysis for JavaScript. In: ISSTA (2013)