Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries

Magnus Madsen
Aarhus University

Benjamin Livshits
Microsoft Research

Michael Fanning
Microsoft Corporation

Microsoft Research Technical Report

MSR-TR-2012-66

Microsoft®
Research

Abstract

JavaScript is a language that is widely-used for both web-based and standalone applications such as those in the Windows 8 operating system. Analysis of JavaScript has long been known to be challenging due to the language's dynamic nature. On top of that, most JavaScript applications rely on large and complex libraries and frameworks, often written in a combination of JavaScript and native code such as C and C++. *Stubs* have been commonly employed as a partial specification mechanism to address the library problem; alas, they are tedious and error-prone.

However, the manner in which library code is *used* within applications often sheds light on what library APIs return or pass into callbacks declared within the application. In this paper, we propose a technique which combines *pointer analysis* with a novel *use analysis* to handle many challenges posed by large JavaScript libraries. Our techniques have been implemented and empirically validated on a set of 25 Windows 8 JavaScript applications, averaging 1,587 lines of code, together with about 30,000 lines of library code, demonstrating a combination of scalability and precision.

I. Introduction

While JavaScript is increasingly used for both web and server-side programming, it is a challenging language for static analysis due to its highly dynamic nature. Recently much attention has been directed at handling the peculiar features of JavaScript in pointer analysis [7, 13, 14], data flow analysis [17, 18], and type systems [25, 26].

The majority of work thus far has largely ignored the fact that JavaScript programs usually execute in a rich execution environment. Indeed, Web applications run in a browser-based environment interacting with the page through the extensive HTML DOM API or sophisticated libraries such as jQuery. Similarly, `node.js` applications run inside an application server. Finally, JavaScript applications in the Windows 8 OS, which are targets of the analysis in this paper, call into the underlying OS through the Windows 8 runtime. In-depth static analysis of these application hinges on our understanding of libraries and frameworks.

Unfortunately, environment libraries such as the HTML DOM and the Windows 8 runtime lack a full JavaScript implementation, relying on underlying C or C++, which is outside of what a JavaScript static analyzers can reason about. Popular libraries, such as jQuery, have a JavaScript implementation, but are hard to reason about due to their heavy use of reflective JavaScript features, such as `eval`, computed properties, runtime addition of fields, etc. The standard solution to overcome these problems is to write partial JavaScript implementations (also known as *stubs*) to partially model library API functionality. Unfortunately, writing stubs is tedious, time-consuming, and error-prone.

Use analysis: The key technical insight that motivates this paper is that observing *uses* of library functionality within application code can shed much light on the structure and functionality of (unavailable) library code. By way of analogy, observing the effect on surrounding planets and stars can provide a way to estimate the mass of an (invisible) black hole. This paper describes how to overcome the challenges described above using an inference approach that combines pointer analysis and *use analysis* to recover necessary information about the structure of objects returned from libraries and passed into callbacks declared within the application.

Example 1 We open with an example illustrating several of the challenges posed by libraries, and how our technique can overcome these challenges. The code below is representative of much DOM-manipulating code that we see in JavaScript applications.

```
var canvas = document.querySelector("#leftcol .logo");
var context = canvas.getContext("2d");
context.fillRect(20, 20, c.width / 2, c.height / 2);
context.strokeRect(0, 0, c.width, c. height);
```

In this example, the call to `querySelector` retrieves a `<canvas>` element represented at runtime by an `HTMLCanvasElement` object; the `context` variable points to a `CanvasRenderingContext2D` object at runtime. `context` is then used for drawing both a filled and stroked rectangle on the `canvas`. Since these objects and functions are implemented as part of the browser API and HTML DOM, no JavaScript implementation that accurately represents them is readily available. Furthermore, note that the return value of the `querySelector` call depends on the CSS expression provided as a string parameter, which is difficult to reason about without an accurate model of the underlying HTML page. There are two common approaches for attempting to analyze this code statically:

- model `querySelector` as (unsoundly) returning a reference to `HTMLElement.prototype`. This approach suffers from a simple problem: `HTMLElement.prototype` does not define `getContext`, so this call will still be unresolved. This is the approach used for auto-completion suggestions in the Eclipse IDE.
- model `querySelector` as returning *any* HTML element within the underlying page. While this approach correctly includes the canvas element, it suffers from returning elements on which `getContext` is undefined. While previous work [17] has focused on tracking elements based on their ids and types, extending this to tracking CSS selector expressions is non-trivial.

Neither solution is really acceptable. In contrast, our analysis will use *pointer information* to resolve the call to `document.querySelector` and then apply *use analysis* to discover that the returned object must have at least three properties: `getContext`, `width`, and `height`, assuming the program runs correctly. Looking through the static heap approximation, only the `HTMLCanvasElement` has all three properties. Assuming we have the whole program available for analysis, this must be the object returned by `querySelector`. From here on, pointer analysis can resolve the remaining calls to `fillRect` and `strokeRect`. □

A. Applications of the Analysis

The idea of combining pointer analysis and use analysis turns out to be powerful and useful in a variety of settings, some of which we outline below.

Call graph discovery: Knowledge about the call graph is useful for a range of analysis tools. Unlike C or Java, in JavaScript call graphs are surprisingly difficult to construct. Reasons for this include reflection, passing anonymous function closures around the program, and lack of static typing, combined with an ad-hoc namespace creation discipline.

API surface discovery: Knowing the portion of an important library such as WinRT utilized by an applications is useful for determining the application's attack perimeter. In aggregate, running this analysis against *many* applications can provide general API use statistics for a library helpful for maintaining it (e.g., by identifying APIs that are commonly used and which may therefore be undesirable to deprecate).

Capability analysis The Windows 8 application model involves a manifest that requires *capabilities* such as `camera_access` or `gps_location_access` to be statically declared. However, in practice, because developers tend to

over-provision capabilities in the manifest [15], static analysis is a useful tool for inferring capabilities that are actually needed [10].

Automatic stub creation Our analysis technique can create stubs from scratch or identify gaps in a given stub collection. When aggregated across a range of application, over time this leads to an enhanced library of stubs useful for analysis and documentation.

Auto-complete Auto-complete or code completion has traditionally been a challenge for JavaScript. Our analysis makes significant strides in the direction of better auto-complete, without resorting to code execution, as shown in Section V-D.

Concrete types Since JavaScript does not have declared types, pointer analysis information is especially useful for computing *concrete types*. Use analysis increases precision by observing how objects are used.

Throughout this paper, our emphasis is on providing a practically useful analysis, favoring practical utility even if it occasionally means sacrificing soundness. Note that our analysis can be a useful building block for (sound) verification tools [**?**]. We further explain the soundness trade-offs in Section II-D.

B. Contributions

Our paper makes the following contributions:

- We propose a strategy for analyzing JavaScript code in the presence of large complex libraries, often implemented in other languages. As a key technical contribution, our analysis combines pointer analysis with a novel *use analysis* that captures how objects returned by and passed into libraries are used within application code, without analyzing library code.
- Our analysis is declarative, expressed as a collection of Datalog inference rules, allowing for easy maintenance, porting, and modification.
- Our techniques in this paper include *partial* and *full* iterative analysis, the former depending on the existence of stubs, the latter analyzing the application without any stubs or library specifications.
- Our analysis is useful for a variety of applications. Use analysis improves points-to results, thereby improving call graph resolution and enabling other important applications such as inferring structured object types, interactive auto-completion, and API use discovery.
- We experimentally evaluate our techniques based on a suite of 25 Windows 8 JavaScript applications, averaging 1,587 line of code, in combination with about 30,000 lines of partial stubs. When using our analysis for call graph resolution, a highly non-trivial task for JavaScript, the median percentage of resolved calls sites increases from 71.5% to 81.5% with partial inference.
- Our analysis is immediately effective in two practical settings. First, our analysis finds about twice as many WinRT API calls compared to a naïve pattern-based analysis (Section V-C). Second, in our auto-completion case study (Section V-D) we out-perform four major widely-used JavaScript IDEs in terms of the quality of auto-complete suggestions.

C. Paper Organization

The rest of the paper is organized as follows. Section II outlines inherent analysis challenges our approach needs to address. Section III gives motivation for the inference techniques this paper uses. Section IV provides a detailed description of our implementation as well as Datalog inference rules. Section V describes our experimental evaluation. Finally, Sections VI and VII describe related work and conclude.

II. Analysis Challenges

Before we proceed further, we summarize the challenges faced by any static analysis tool when trying to analyze JavaScript applications that depend on libraries.

A. Whole Program Analysis

Whole program analysis in JavaScript has long been known to be problematic [7, 14]. Indeed, libraries such as the Browser API, the HTML DOM, `node.js` and the Windows 8 API are all implemented in native languages such as C and C++; these implementations are therefore often simply unavailable to static analysis. Since no JavaScript implementation exists, static analysis tool authors are often forced to create stubs. This, however, brings in the issues of stub completeness as well as development costs. Finally, JavaScript code frequently uses dynamic code loading, requiring static analysis at runtime [14], further complicating whole-program analysis.

B. Underlying Libraries & Frameworks

While analyzing code that relies on rich libraries has been recognized as a challenge for languages such as Java, JavaScript presents a set of unique issues.

Complexity: Even if the application code is well-behaved and amenable to analysis, complex JavaScript applications frequently use libraries such as jQuery and Prototype. While these are implemented in JavaScript, they present their own challenges because of extensive use of reflection such as `eval` or computed property names. Recent work has made some progress towards understanding and handling `eval` [16, 23], but these approaches are still fairly limited and do not fully handle all the challenges inherent to large applications.

Scale of libraries: Underlying libraries and frameworks are often very large. In the case of Windows 8 applications, they are around 30,000 lines of code, compared to 1,587 for applications on average. Requiring them to be analyzed every time an application is subjected to analysis results in excessively long running times for the static analyzer.

C. Tracking Interprocedural Flow

Points-to analysis effectively embeds an analysis of interprocedural data flow to model how data is copied across the program. However, properly modeling interprocedural data flow is a formidable task.

Containers: The use of arrays, lists, maps, and other complex data structures frequently leads to conflated data flow in static analysis; an example of this is when analysis is not able to statically differentiate distinct indices of an array. This problem is exacerbated in JavaScript because of excessive use of the DOM, which can be addressed both directly and through tree pointer traversal. For instance `document.body` is a direct lookup, whereas `document.getElementsByName("body")[0]` is an indirect lookup. Such indirect lookups present special challenges for static analyses because they require explicit tracking of the association between lookup keys and their values. This problem quickly becomes unmanageable when CSS selector expressions are considered (e.g., the jQuery `$()` selector function), as this would require the static analysis to reason about the tree structure of the page.

Reflective calls: Another typical challenge of analyzing JavaScript code stems from reflective calls into application code being "invisible" [20]. As a result, callbacks within the application invoked reflectively will have no actuals linked to their formal parameters, leading to variables within these callback functions having empty points-to sets.

D. Soundness

While soundness is a highly attractive goal for static analysis, it is not one we are pursuing in this paper, opting for practical utility on a range of applications that do not necessarily require soundness. JavaScript is a complex language with a number of dynamic features that are difficult or impossible to handle fully statically [16, 23] and hard-to-understand semantic features [9, 21].

We have considered several options before making the decision to forgo conservative analysis. One approach is defining language subsets to capture when analysis results are sound. While this is a well-known strategy, we believe that following this path leads to unsatisfactory results.

First, language restrictions are too strict for real programs: dealing with difficult-to-analyze JavaScript language constructs such as with and `eval` and intricacies of the execution environment such as the Function object, etc. While language restrictions have been used in the past [13], these limitations generally only apply to small programs (i.e. Google Widgets, which are mostly under 100 lines of code). When one is faced with bodies of code consisting of thousands of lines and containing complex libraries such as jQuery, most language restrictions are immediately violated.

Second, soundness assumptions are too difficult to understand and verify statically. For example, in Ali *et al.* [1] merely *explaining* the soundness assumptions requires three pages of text. It is not clear how to *statically* check these assumptions, either, not to mention doing so efficiently, bringing the practical utility of such an approach into question.

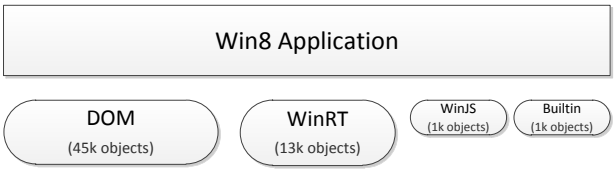


Fig. 1: Composition of a typical Windows 8 JavaScript application.

Name	Lines	Functions	Alloc. sites	Fields
Builtin	225	161	1,039	190
DOM	21,881	12,696	44,947	1,326
WinJS	404	346	1,114	445
Windows 8 API	7,213	2,970	13,989	3,834
Total	29,723	16,173	61,089	5,795

Fig. 2: Approximate stub sizes for widely-used libraries.

Example 2 Consider the code below as an illustration of why pursuing a fully sound approach will yield results that unusable in practice:

```
var x = getLibraryObject();
if (P) {
    y = x.foo();
} else {
    y = x.toString();
}
```

indeed, in this case, a conservative analysis will conclude that `x` may have function `toString` on it, unifying `x` with *all* objects, whereas our analysis will flow-insensitively conclude that it must have both `toString` and `foo`, only unifying `x` with (fewer) objects that have function `foo` available. □

III. Overview

The composition of a Windows 8 (or Win8) JavaScript application is illustrated in Figure 1. These are frequently complex applications that are not built in isolation: in addition to resources such as images and HTML, Win8 applications depend on a range of JavaScript libraries for communicating with the DOM, both using the built-in JavaScript DOM API and rich libraries such as jQuery and WinJS (an application framework and collection of supporting APIs used for Windows 8 HTML development), as well as the underlying Windows runtime. To provide a sense of scale, information about commonly used stub collections is shown in Figure 2.

A. Analysis Overview

The intuition for the work in this paper is that despite having incomplete information about this extensive library functionality, we can still discern much from observing how developers use library code. For example, if there is a call whose base is a variable obtained from a library, the variable must refer to a function for the call to succeed. Similarly, if there is a load whose base is a variable returned from a library call, the variable must refer to an object that has that property for the load to succeed.

Example 3 A summary of the connection between the concrete pointer analysis and use analysis described in this paper is graphically illustrated in Figure 3. In this example, function `process` invokes functions `mute` and `playSound`, depending on which button has been pressed. Both callees accept variable `a`, an alias of a library-defined `Windows.Media.Audio`, as a parameter. The arrows in the figure represent the flow of constraints.

Points-to analysis (downward arrows) flows facts from actuals to formals — functions receive information about the arguments passed into them, while the *use analysis* (upward arrows) works in the opposite direction, flowing *demands* on the shape of objects passed from formals to actuals.

Specifically, the points-to analysis flows variable `a`, defined in `process`, to formals `x` and `y`. Within functions `playSound` and `mute` we discover that these formal arguments must have functions `Volume` and `Mute` defined on them, which flows back to the library object that variable `a` must point to. Its shape as a result must contain at least functions `Volume` and `Mute`. □

Use analysis: The notion of use analysis above leads us to an inference technique, which comes in two flavors: *partial* and *full*.

Partial inference assumes that stubs for libraries are available. Stubs are not required to be complete implementations, instead, function implementations are frequently completely omitted, leading to missing data flow. What is required is that all objects, functions and properties (JavaScript term for *fields*) exposed by the library are described in the stub. Partial inference solves the problem of missing flow between library and application code by linking together objects of matching shapes, a process we call *unification* (Section IV-C).

Full inference is similar to partial inference, but goes further in that it does not depend on the existence of any stubs. Instead it attempts to infer library APIs based on uses found in the application. Paradoxically, full inference is often faster than partial inference, as it does not need to analyze large collections of stubs, which is also wasteful, as a typical application only requires a small portion of them.

In the rest of this section, we will build up the intuition for the analysis we formulate. Precise analysis details are found in Section IV.

Library stubs: Stubs are commonly used for static analysis in a variety of languages, starting from `libc` stubs for C programs, to complex and numerous stubs for JavaScript built-ins and DOM functionality.

Example 4 Here is an example of stubs from the WinRT library. Note that stub functions are empty.

```
Windows.Storage.Stream.FileOutputStream = function() {};
Windows.Storage.Strean.FileOutputStream.prototype = {
    writeAsync = function() {},
    flushAsync = function() {},
    close = function() {}
}
```

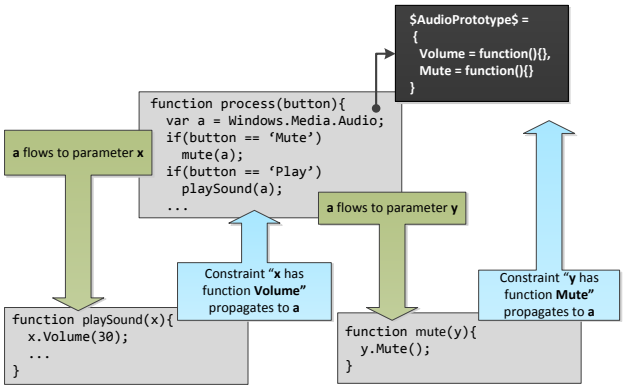


Fig. 3: Points-to flowing facts downwards and use analysis flowing data upwards.

This stub models the structure of the `FileOuputStream` object and its `prototype` object. It does not, however, capture the fact that `writeAsync` and `flushAsync` functions return an `AsyncResults` object. Use analysis can, however, discover this if we consider the following code:

```
var s = Windows.Storage.Stream;
var fs = new s.FileOutputStream(...)
fs.writeAsync(...).then(function() {
  ...
});
```

We can observe from this that `fs.writeAsync` should return an object whose `then` is a function. These facts allow us to unify the return result of `writeAsync` with the `Promise`$[\![$`Proto`$]\!]$ object, the prototype of the `Promise` object declared in the WinJS library. □

B. Symbolic Locations and Unification

Abstract locations are typically used in program analyses such as a points-to analysis to approximate objects allocated in the program at runtime. We employ the allocation site abstraction as an approximation of runtime object allocation (denoted by domain H in our analysis formulation). In this paper we consider the partial and full inference scenarios.

It is useful to distinguish between abstract locations in the heap within the application (denoted as H_A) and those within libraries (denoted as H_L). Additionally, we maintain a set of *symbolic locations* H_S; these are necessary for reasoning about results returned by library calls. In general, both library and application abstract locations may be returned from such a call.

It is instructive to consider the connections between the variable V and heap H domains. Figure 4a shows a connection between variables and the heap $H = H_A \cup H_S \cup H_L$ in the context of partial inference. Figure 4b shows a similar connection between variables and the heap $H = H_A \cup H_S$ in the context of full inference, which lacks H_L. Variables within the V domain have points-to links to heap elements in H; H elements are connected with points-to links that have property names.

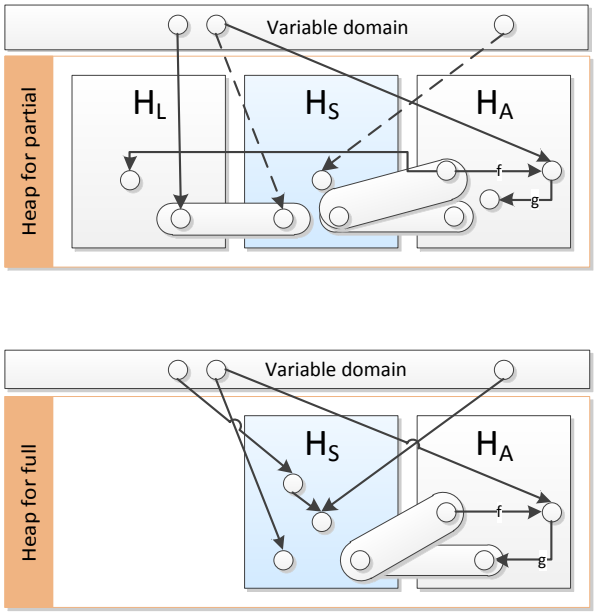


Fig. 4: Partial (a, above) and full (b, below) heaps.

Since at runtime actual objects are either allocated within the application (H_A) or library code (H_L), we need to unify symbolic locations H_S with those in H_A and H_L.

C. Inference Algorithm

Because of missing interprocedural flow, a fundamental problem with building a practical and usable points-to analysis is that sometimes variables do not have any abstract locations that they may point to. Of course, with the exception of dead code or variables that only point to `null` and `undefined`, this is a static analysis artifact. In the presence of libraries, several distinct scenarios lead to

- **dead returns**: when a library function stub lacks a return value;
- **dead arguments**: when a callback within the application is passed into a library and the library stub fails to properly invoke the callback;
- **dead loads**: when the base object reference (receiver) has no points-to targets.

Strategy: Our overall strategy is to create symbolic locations for all the scenarios above. We implement an iterative algorithm. At each iteration, we run a points-to analysis pass and then proceed to collect dead arguments, returns, and loads, introducing symbol locations for each. We then perform a unification step, where symbolic locations are unified with abstract locations. A detailed description of this process is given in Section IV.

Iterative solution: An iterative process is required because we may discover new points-to targets in the process of unification. As the points-to relation grows, additional dead arguments, returns, or loads are generally discovered, requiring further iterations. Iteration is terminated when we can no longer find dead arguments, dead returns, or dead loads, and no more unification is possible. Note that the only algorithmic change for full analysis is the need to create symbolic locations for dead loads. We evaluate the iterative behavior experimentally in Section V-E.

NEWOBJ$(v_1 : V, h : H, v_2 : V)$	object instantiation
ASSIGN$(v_1 : V, v_2 : V)$	variable assignment
LOAD$(v_1 : V, v_2 : V, p : P)$	property load
STORE$(v_1 : V, p : P, v_2 : V)$	property store
FORMALARG$(f : H, z : Z, v : V)$	formal argument
FORMALRET$(f : H, v : V)$	formal return
ACTUALARG$(c : C, z : Z, v : V)$	actual argument
ACTUALRET$(c : C, v : V)$	actual return
CALLGRAPH$(c : C, f : H)$	indicates that f may be invoked by a call site c
POINTSTO$(v : V, h : H)$	indicates that v may point to h
HEAPPTSTO$(h_1 : H, p : P, h_2 : H)$	indicates that h_1's p property may point to h_2
PROTOTYPE$(h_1 : H, h_2 : H)$	indicates that h_1 may have h_2 in its internal prototype chain

Fig. 5: Datalog relations used for program representation and pointer analysis.

Unification strategies: Unification is the process of linking or matching symbolic locations with matching abstract locations. In Section IV-C, we explore three strategies: unify based on matching of a single property, all properties, and prototype-based unification.

IV. TECHNIQUES

We base our technique on pointer analysis and use analysis. The pointer-analysis is a relatively straightforward flow- and context-insensitive subset-based analysis described in Guarnieri *et al.* [13]. The analysis is field-sensitive, meaning that it distinguishes properties of different abstract objects. The call-graph is constructed on-the-fly because JavaScript has higher-order functions, and so the points-to and call graph relations are mutually dependent. The use analysis is based

APPALLOC$(h : H)$, APPVAR$(v : V)$	represents that the allocation site or variable originates from the application and not the library
SPECIALPROPERTY$(p : P)$	properties with special semantics or common properties, such as `prototype` or `length`
PROTOTYPEOBJ$(h : H)$	indicates that the object is used as a prototype object

Fig. 6: Additional Datalog facts for use analysis.

$$
\begin{array}{lll}
\textsc{PointsTo}(v, h) & :- & \textsc{NewObj}(v, h, \_). \\
\textsc{PointsTo}(v_1, h) & :- & \textsc{Assign}(v_1, v_2), \textsc{PointsTo}(v_2, h). \\
\textsc{PointsTo}(v_2, h_2) & :- & \textsc{Load}(v_2, v_1, p), \textsc{PointsTo}(v_1, h_1), \textsc{HeapPtsTo}(h_1, p, h_2). \\
 & & \\
\textsc{HeapPtsTo}(h_1, p, h_2) & :- & \textsc{Store}(v_1, p, v_2), \textsc{PointsTo}(v_1, h_2), \textsc{PointsTo}(v_2, h_2). \\
\textsc{HeapPtsTo}(h_1, p, h_3) & :- & \textsc{Prototype}(h_1, h_2), \textsc{HeapPtsTo}(h_2, p, h_3). \\
\textsc{Prototype}(h_1, h_2) & :- & \textsc{NewObj}(\_, h_1, v), \textsc{PointsTo}(v, f), \textsc{HeapPtsTo}(f, \texttt{"prototype"}, h_3). \\
 & & \\
\textsc{CallGraph}(c, f) & :- & \textsc{ActualArg}(c, 0, v), \textsc{PointsTo}(v, f). \\
\textsc{Assign}(v_1, v_2) & :- & \textsc{CallGraph}(c, f), \textsc{FormalArg}(f, i, v_1), \textsc{ActualArg}(c, i, v_2), z > 0. \\
\textsc{Assign}(v_2, v_1) & :- & \textsc{CallGraph}(c, f), \textsc{FormalRet}(f, v_1), \textsc{ActualRet}(c, v_2).
\end{array}
$$

(a) Inference rules for an inclusion-based points-to analysis expressed in Datalog.

$$
\begin{array}{lll}
\textsc{ResolvedVariable}(v) & :- & \textsc{PointsTo}(v, \_). \\
\textsc{PrototypeObj}(h) & :- & \textsc{Prototype}(\_, h). \\
 & & \\
\textsc{DeadArgument}(f, i) & :- & \textsc{FormalArg}(f, i, v), \neg\textsc{ResolvedVariable}(v), \textsc{AppAlloc}(f), i > 1. \\
\textsc{DeadReturn}(c, v_2) & :- & \textsc{ActualArg}(c, 0, v_1), \textsc{PointsTo}(v_1, f), \textsc{ActualRet}(c, v_2), \\
 & & \neg\textsc{ResolvedVariable}(v_2), \neg\textsc{AppAlloc}(f). \\
 & & \\
\textsc{DeadLoad}(h, p) & :- & \textsc{Load}(v_1, v_2, p), \textsc{PointsTo}(v_2, h), \neg\textsc{HasProperty}(h, p), \textsc{AppVar}(v_1), \textsc{AppVar}(v_2). \\
\textsc{DeadLoad}(h_2, p) & :- & \textsc{Load}(v_1, v_2, p), \textsc{PointsTo}(v_2, h_1), \textsc{Prototype}(h_1, h_2), \\
 & & \neg\textsc{HasProperty}(h_2, p), \textsc{Symbolic}(h_2), \textsc{AppVar}(v_1), \textsc{AppVar}(v_2). \\
\textsc{DeadLoadDynamic}(v_1, h) & :- & \textsc{LoadDynamic}(v_1, v_2), \textsc{PointsTo}(v_2, h), \neg\textsc{ResolvedVariable}(v_1), \\
 & & \textsc{AppVar}(v_1), \textsc{AppVar}(v_2). \\
 & & \\
\textsc{DeadPrototype}(h_1) & :- & \textsc{NewObj}(\_, h, v), \textsc{PointsTo}(v, f), \textsc{Symbolic}(f), \neg\textsc{HasSymbolicPrototype}(h). \\
 & & \\
\textsc{CandidateObject}(h_1, h_2) & :- & \textsc{DeadLoad}(h_1, p), \textsc{HasProperty}(h_2, p), \textsc{Symbolic}(h_1), \neg\textsc{Symbolic}(h_2), \\
 & & \neg\textsc{HasDynamicProps}(h_1), \neg\textsc{HasDynamicProps}(h_2), \neg\textsc{SpecialProperty}(p). \\
\textsc{CandidateProto}(h_1, h_2) & :- & \textsc{DeadLoad}(h_1, p), \textsc{HasProperty}(h_2, p), \textsc{Symbolic}(h_1), \neg\textsc{Symbolic}(h_2), \\
 & & \neg\textsc{HasDynamicProps}(h_1), \neg\textsc{HasDynamicProps}(h_2), \textsc{PrototypeObj}(h_2). \\
 & & \\
\textsc{NoLocalMatch}(h_1, h_2) & :- & \textsc{Prototype}(h_2, h_3), \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_2, p), \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_3, p), \\
 & & \textsc{CandidateProto}(h_1, h_2), \textsc{CandidateProto}(h_1, h_3), h_2 \neq h_3. \\
 & & \\
\textsc{UnifyProto}(h_1, h_2) & :- & \neg\textsc{NoLocalMatch}(h_1, h_2), \textsc{CandidateProto}(h_1, h_2). \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_2, p). \\
\textsc{FoundPrototypeMatch}(h) & :- & \textsc{UnifyProto}(h, \_). \\
\textsc{UnifyObject}(h_1, h_2) & :- & \textsc{CandidateObject}(h_1, h_2), \neg\textsc{FoundPrototypeMatch}(h_1) \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_2, p).
\end{array}
$$

(b) Use analysis inference.

Fig. 7: Inference rules for both points-to and use analysis.

on unification of symbolic and abstract locations based on property names.

A. Pointer Analysis

The input program is represented as a set of facts in relations of fixed arity and type summarized in Figure 5 and described below. Relations use the following *domains*: heap-allocated objects and functions H, program variables V, call sites C, properties P, and integers Z.

The pointer analysis implementation is formulated declaratively using Datalog, as has been done in range of prior projects such as Whaley for Java *et al.* [27] and Gatekeeper for JavaScript [13]. The JavaScript application is first normalized and then converted into a set of facts. These are combined with Datalog analysis rules resolved using the Microsoft Z3 fixpoint solver [8].

Rules for the Andersen-style inclusion-based [2] points-to analysis are shown in Figure 7a. In the rest of this section, we shall use the $\forall$ quantifier and $\Rightarrow$ implication connectives in our Datalog rules to ease presentation. While these connectives are usually not supported in Datalog engines, they can be encoded as follows: $C(x, z) :- \forall y.A(x, y) \Rightarrow B(y, z).$ is equivalent to the Datalog rules:

$$
\begin{array}{lll}
N(x, z) & :- & A(x, y), \neg B(y, z). \\
C(x, z) & :- & A(x, y), B(y, z), \neg N(x, z).
\end{array}
$$

B. Extending with Partial Inference

We now describe how the basic pointer analysis can be extended with use analysis in the form of *partial inference*. In partial inference we assume the existence of stubs that describe all objects, functions and properties. Function implementations, as stated before, may be omitted. The purpose of partial inference is to recover missing *flow* due to missing implementations. Flow may be missing in three different places: arguments, return values, and loads.

- $\textsc{DeadLoad}(h : H, p : P)$ where h is an abstract location and p is a property name. Records that property p is accessed from h, but h lacks a p property. We capture this with the rule:

$$
\begin{array}{lll}
\textsc{DeadLoad}(h, p) & :- & \textsc{Load}(v_1, v_2, p), \\
 & & \textsc{PointsTo}(v_2, h), \\
 & & \neg\textsc{HasProperty}(h, p), \\
 & & \textsc{AppVar}(v_1), \\
 & & \textsc{AppVar}(v_2).
\end{array}
$$

  Here the $\textsc{PointsTo}(v_2, h)$ constraint ensures that the base object is resolved. The two $\textsc{AppVar}$ constraints ensure that

the load actually occurs in the application code and not the library code.

- DEADARGUMENT$(f : H, i : Z)$ where f is a function and i is an argument index. Records that the i'th argument has no value. We capture this with the rule:

$$\begin{aligned}\text{DEADARGUMENT}(f, i) \quad :- \quad & \text{FORMALARG}(f, i, v),\\ & \neg\text{RESOLVEDVARIABLE}(v),\\ & \text{APPALLOC}(f),\\ & z > 1.\end{aligned}$$

  Here the APPALLOC constraint ensures that the argument occurs in a function within the application code, and not in the library code; argument counting starts at 1.
- DEADRETURN$(c : C, v : V)$ where c is a call site and v is the result value. Records that the return value for call site c has no value.

$$\begin{aligned}\text{DEADRETURN}(c, v_2) \quad :- \quad & \text{ACTUALARG}(i, 0, v_1),\\ & \text{POINTSTO}(v_1, f),\\ & \text{ACTUALRET}(i, v_2),\\ & \neg\text{RESOLVEDVARIABLE}(v_2),\\ & \neg\text{APPALLOC}(f).\end{aligned}$$

  Here the POINTSTO(v_1, f) constraint ensures that the call site has call targets. The $\neg$APPALLOC(f) constraint ensures that the function called is not an application function, but either (a) a library function or (b) a symbolic location.

We use these relations to introduce symbolic locations into POINTSTO, HEAPPTSTO, and PROTOTYPE as shown in Figure 8. In particular for every dead load, dead argument and dead return we introduce a fresh symbolic location. We restrict the introduction of dead loads by requiring that the base object is not a symbolic object, unless we are operating in full inference mode. This means that every load must be unified with an abstract object, before we consider further unification for properties on that object. In full inference we have to drop this restriction, because not all objects are known to the analysis.

C. Unification

Unification is the process of linking or matching symbolic locations s with matching abstract locations l. The simplest form of unification is to do no unification at all. In this case no actual flow is recovered in the application. Below we explore unification strategies based on property names.

$\exists$ **shared properties:** The obvious choice here is to link objects which share at least one property. Unfortunately, with this strategy, most objects quickly become linked. Especially problematic are properties with common names, such as `length` or `toString`, as all objects have these properties.

$\forall$ **shared properties:** We can improve upon this strategy by requiring that the linked object must have *all* properties of the symbolic object. This drastically cuts down the amount of unification, but because the shape of s is an over-approximation, requiring all properties to be present may link to too few objects, introducing unsoundness. It can also introduce imprecision: if we have s with function `trim()`, we will unify s to all string constants in the program. The following rule

INFERENCE$(constraints, facts, isFull)$
1 $relations$ = SOLVE-CONSTRAINTS$(constraints, facts)$
2 **repeat**
3 $\quad newFacts$ = MAKE-SYMBOLS$(relations, isFull)$
4 $\quad facts = facts \cup newFacts$
5 $\quad relations$ = SOLVE-CONSTRAINTS$(constraints, facts)$
6 **until** $newFacts$ == $\emptyset$

MAKE-SYMBOLS$(relations, isFull)$
1 $facts = \emptyset$
2 **for** $(h, p) \in relations.$DEADLOAD $: H \times P$
3 $\quad$ **if** $\neg$SYMBOLIC(h) or $isFull$
4 $\quad\quad facts_{\cup}$ = `new` HEAPPTSTO$(h, p,$ `new` $H)$
5 **for** $(f, i) \in relations.$DEADARGUMENT $: H \times Z$
6 $\quad v$ = FORMALARG$[f, i]$
7 $\quad facts_{\cup}$ = `new` POINTSTO$(v,$ `new` $H)$
8 **for** $(c, v) \in relations.$DEADRETURN $: C \times V$
9 $\quad facts_{\cup}$ = `new` POINTSTO$(v,$ `new` $H)$
10 // Unification:
11 **for** $h \in relations.$DEADPROTOTYPE $: H$
12 $\quad facts_{\cup}$ = `new` PROTOTYPE$(h,$ `new` $H)$
13 **for** $(h_1, h_2) \in relations.$UNIFYPROTO $: H \times H$
14 $\quad facts_{\cup}$ = `new` PROTOTYPE(h_1, h_2)
15 **for** $(h_1, h_2) \in relations.$UNIFYOBJECT $: H \times H$
16 $\quad$ **for** $(h_3, p, h_1) \in relations.$HEAPPTSTO $: H \times P \times H$
17 $\quad\quad facts_{\cup}$ = `new` HEAPPTSTO(h_3, p, h_2)
18 **return** $facts$

Fig. 8: Iterative inference algorithms.

$$\begin{aligned}\text{CANDIDATEOBJECT}(h_1, h_2) \quad :- \quad & \text{DEADLOAD}(h_1, p),\\ & \text{HASPROPERTY}(h_2, p),\\ & \text{SYMBOLIC}(h_1),\\ & \neg\text{SYMBOLIC}(h_2),\\ & \neg\text{HASDYNAMICPROPS}(h_1),\\ & \neg\text{HASDYNAMICPROPS}(h_2),\\ & \neg\text{SPECIALPROPERTY}(p).\end{aligned}$$

expresses which symbolic and abstract locations h_1 and h_2 are *candidates* for unification. First, we require that the symbolic and abstract location share at least one property. Second, we require that neither the symbolic nor the abstract object have *dynamic* properties. Third, we disallow commonly-used properties, such as `prototype` and `length`, as evidence for unification. The relation below captures when two locations h_1 and h_2 are unified:

$$\begin{aligned}\text{UNIFYOBJECT}(h_1, h_2) \quad :- \quad & \text{CANDIDATEOBJECT}(h_1, h_2),\\ & \forall p.\text{DEADLOAD}(h_1, p) \Rightarrow\\ & \quad \text{HASPROPERTY}(h_2, p).\end{aligned}$$

This states that h_1 and h_2 must be candidates for unification and that if a property p is accessed from h_1 then that property must be present on h_2. If h_1 and h_2 are unified then the HEAPPTSTO relation is extended such that any place where h_1 may occur h_2 may now also occur.

Prototype-based unification: Instead of attempting to unify with all possible abstract locations l, an often better strategy is to only unify with those that serve as prototype objects. Such objects are used in a two-step unification procedure: first, we

see if all properties of a symbolic object can be satisfied by a prototype object, if so we unify them and stop the procedure. If not, we consider all non-prototype objects. We take the prototype hierarchy into consideration by unifying with the most precise prototype object.

Example 5 The following example illustrates how this can improve precision:

```
var firstName     = "Lucky";
var lastName      = "Luke";
var favoriteHorse = "Jolly Jumper";
function compareIgnoreCase(s1, s2) {
  return s1.toLowerCase() < s2.toLowerCase();
}
```

Here we have three string constants and a comparator function. Assume that the comparator is passed into a library as a callback. In this case the pointer analysis does not know what the two arguments `s1` and `s2` may point to, but the use analysis knows that these arguments must have a `toLowerCase` property. The unification, described so far, would continue by linking the arguments to all abstract locations which have the `toLowerCase` property.

Unfortunately, all string constants have this property, so this over-approximation is overly imprecise. We obtain better unification by first considering prototype objects. In this case we discover that the `String⟦Proto⟧` object has the `toLowerCase` property. In prototype-based unification, we merely conclude that the prototype of `s1` and `s2` must be `String⟦Proto⟧`. □

In the above discussion we did not precisely define what we consider to be prototype objects: we consider all objects which may flow to the `prototype` property of some object to be prototype objects. Furthermore built-in prototype objects, such as `Array⟦Proto⟧` and `String⟦Proto⟧`, are known to be prototype objects. This is captured by the PROTOTYPEOBJ rule.

One issue remains: What if multiple prototype objects are possible for unification? In this case we select the most precise object in the prototype hierarchy, i.e. the object that is *highest* in the prototype chain. This rule captures the fact that it is possible to unify h_1 with h_2, but there is also some h_3 in the prototype chain of h_2 that could be unified with h_1. This means that h_1 and h_2 should not be unified.

$$\begin{aligned}\textsc{NoLocalMatch}(h_1, h_2) :- \\ &\textsc{Prototype}(h_2, h_3), \\ &\forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_2, p), \\ &\forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_3, p), \\ &\textsc{CandidateProto}(h_1, h_2), \\ &\textsc{CandidateProto}(h_1, h_3), \\ &h_2 \neq h_3.\end{aligned}$$

We can define prototype-based unification as

$$\begin{aligned}\textsc{UnifyProto}(h_1, h_2) :- \\ &\neg\textsc{NoLocalMatch}(h_1, h_2), \\ &\textsc{CandidateProto}(h_1, h_2). \\ &\forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_2, p).\end{aligned}$$

The above captures that h_1 and h_2 are compatible and there is no matching object in the prototype chain of h_2.

D. Extending with Full Inference

As shown in the pseudo-code in Figure 8, we can extend the analysis to support full inference with a simple change. Recall, in full inference we do not assume the existence of any stubs, and the application is analyzed completely by itself. We implement this by dropping the restriction that symbolic locations are only introduced for non-symbolic locations. Instead we will allow a property of a symbolic location to point to another symbolic location.

Introducing these symbolic locations will resolve a load, and in doing so potentially resolve the base of another load. This is in turn may cause another dead load to appear for that base object. In this way the algorithm can be viewed as a frontier expansion along the known base objects. At each iteration the frontier is expanded by one level. This process cannot go on forever, as there is only a fixed number of loads, and thereby dead loads, and at each iteration at least one dead load is resolved.

E. Namespace Mechanisms

JavaScript has no built-in namespace mechanism and lacks features for proper encapsulation. Thus, many libraries provide functionality for emulating these features. A common approach is to have a function which takes a string (the namespace) and an object literal, and then creates a namespace of the shape captured by the object literal:

Example 6 Namespace creation.

```
WinJS.Namespace.define("GameManager", {
    scoreHelper: new ScoreHelper()
});
GameManager.scoreHelper.newScore(...);
```

This, of course, presents a challenge to static analysis. Luckily, based on the shape information, use analysis and unification can infer that the global variable `GameManager` points to the object literal passed into the `define` function. As a result, we are able to resolve the call to `newScore`. While unification succeeds for the above example, some namespace mechanisms require more work:

```
var AssetManager = WinJS.Class.define(null, {
    playSound: function (sound, volume) { ... }
});
var assetManager = new AssetManager();
assertManager.playSound();
```

Here the namespace mechanism is used to create a constructor. The prototype of this function is set to the object literal passed into `define`. Of course, the analysis has no way of knowing this, since no implementation is available. □

We remedy situations like the one described above by considering objects with *dead prototypes*. We introduce

$$\begin{aligned}\textsc{DeadPrototype}(h_1) \quad :- \quad &\textsc{NewObj}(\_, h, v), \\ &\textsc{PointsTo}(v, f), \\ &\textsc{Symbolic}(f), \\ &\neg\textsc{HasSymbolicPrototype}(h).\end{aligned}$$

to track objects that may have missing prototypes. As shown in Figure 8, at each analysis iteration, for every dead prototype, we introduce a symbolic object into the prototype chain. This symbolic object is then subject to unification like all other symbolic objects. One issue remains: this symbolic prototype object is never considered in the definition of DEADLOAD and so is not subject to unification. We handle this by allowing use analysis to propagate dead loads upwards in the prototype chain:

$$\begin{aligned}\textsc{DeadLoad}(h_2, p) \quad :- \quad & \textsc{Load}(v_1, v_2, p),\\ & \textsc{PointsTo}(v_2, h_1),\\ & \textsc{Prototype}(h_1, h_2),\\ & \neg\textsc{HasProperty}(h_2, p),\\ & \textsc{Symbolic}(h_2),\\ & \textsc{AppVar}(v_1),\\ & \textsc{AppVar}(v_2).\end{aligned}$$

Applied to the example above, the analysis will introduce a symbolic prototype object for the `new` statement. The (unresolved) access to `playSound` is then propagated up the prototype chain. Thus the access to `playSound` becomes a dead load on the symbolic prototype object. This in turn allows unification of the symbolic object and the object literal. Finally, the call to `playSound` is properly resolved using the newly unified object in the prototype chain.

F. Array Access and Dynamic Properties

We now show how our technique can be extended to deal with array accesses and computed (non-static) property names. We refer to such loads as *dynamic loads*. Unfortunately, reasoning about such loads precisely usually requires more expensive analysis features, such as flow sensitivity and analysis of symbolic expressions. As we do not wish to extend our analysis with these techniques, we opt for a light-weight approach. We extend the analysis to track dead dynamic loads:

- $\textsc{DeadLoadDynamic}(v, h)$ where v is a variable and h is an abstract location, states that there is load of an unknown property from h and the result of this load flows into v.

Unlike regular DEADLOADS we do not wish to introduce a symbolic object for every possible property of H, instead we introduce a single symbolic object and inject it directly into V. This creates a "disconnect" in the heap, but allows the use analysis to proceed.

Example 7 Here is an example demonstrating this:

```
function copyAll(files, ext, toDir) {
  for(var i = 0; i < files.length; i++) {
    var f = files[i];
    if (f.FileType == ext) {
      f.CopyAsync(toDir);
    }
  }
}
```

The dynamic load occurs within expression `files[i]`. Our analysis does not track integers and conservatively believes that any property could be read from `files`. This in turn causes the use analysis to conclude that `files` may have all possible properties. Usually, this will cause unification to fail and never find any abstract locations to unify with.

We deal with this situation by introducing a *single* special symbolic location. This symbolic location is never unified with anything and so is left unresolved, but it does allow the use-analysis to continue for properties read from that symbolic object. In the above example, this special symbolic location is introduced for `files` and the use analysis proceeds to discover that `f` must have properties `FileType` and `CopyAsync`. This information is then used for unification of `f` with `StorageFile`. □

Lines	Functions	Alloc. sites	Call sites	Properties	Variables
245	11	128	113	231	470
345	74	606	345	298	1,749
402	27	236	137	298	769
434	51	282	194	336	1,007
488	53	369	216	303	1,102
627	59	341	239	353	1,230
647	36	634	175	477	1,333
711	315	1,806	827	670	5,038
735	66	457	242	363	1,567
807	70	467	287	354	1,600
827	33	357	149	315	1,370
843	63	532	268	390	1,704
1,010	138	945	614	451	3,223
1,079	84	989	722	396	2,873
1,088	64	716	266	446	2,394
1,106	119	793	424	413	2,482
1,856	137	991	563	490	3,347
2,141	209	2,238	1,354	428	6,839
2,351	192	1,537	801	525	4,412
2,524	228	1,712	1,203	552	5,321
3,159	161	2,335	799	641	7,326
3,189	244	2,333	939	534	6,297
3,243	108	1,654	740	515	4,517
3,638	305	2,529	1,153	537	7,139
6,169	506	3,682	2,994	725	12,667
1,587	**134**	**1,147**	**631**	**442**	**3,511**

Fig. 9: Benchmarks, sorted by lines of code.

V. EXPERIMENTAL EVALUATION

This section talks about our experimental setup in Section V-A. Call graph resolution is discussed in Section V-B. Sections V-C and V-D presents case studies of WinRT API use inference and auto-completion. Finally, Section V-E talks about analysis performance.

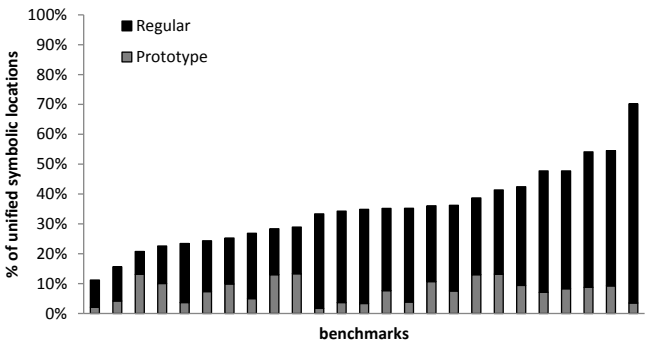


Fig. 10: % of unified symbolic locations for partial inference. Gray is prototype unification. Black is regular unification. Sorted by total unification percentage.

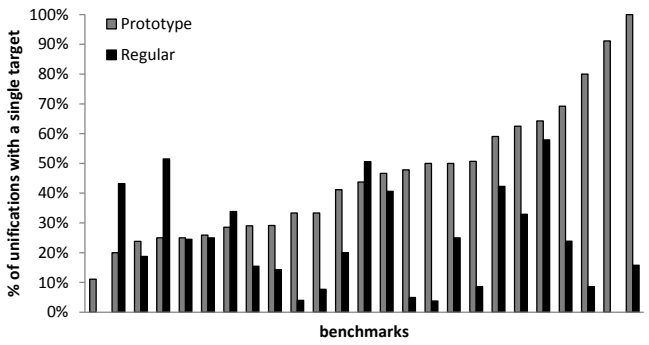


Fig. 11: % of unifications which have a single target. Gray is prototype unification. Black is regular unification. Sorted by the number of prototype unifications with a single target.

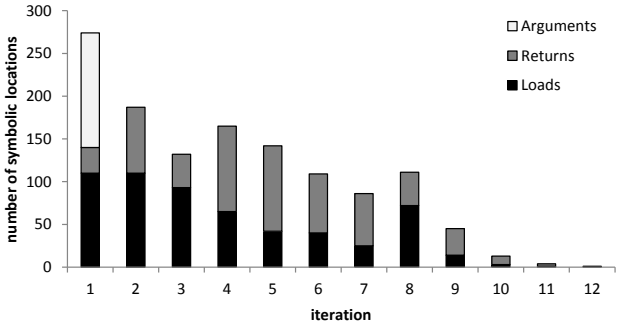
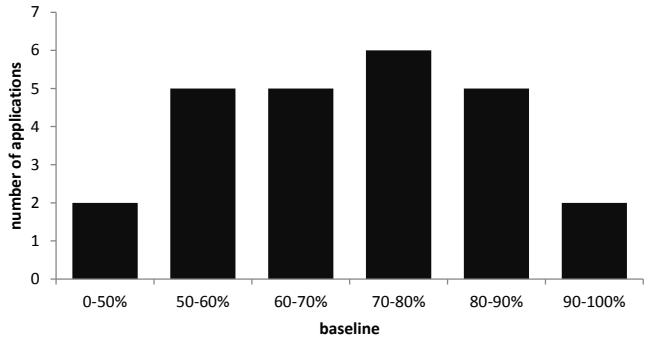


Fig. 12: Progression of full inference on a single benchmark.



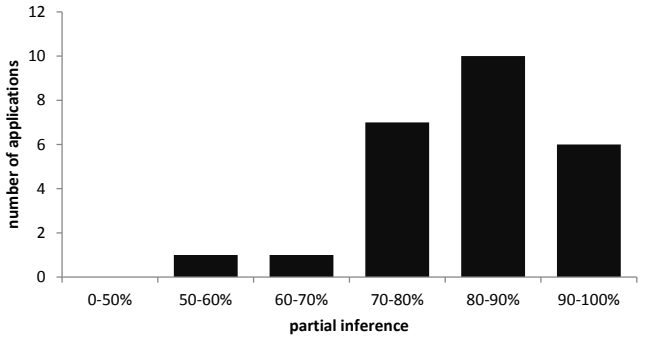


Fig. 13: % of resolved call sites for baseline (points-to without stubs) and partial inference.

A. Experimental Setup

We have implemented both the *partial* and *full* inference techniques described in this paper. Our tool is split into a front-end written in C# and a back-end which uses analysis rules encoded in Datalog, as shown in Section IV. The front-end parses JavaScript application files and library stubs and generates input facts from them. The back-end iteratively executes the Z3 Datalog engine [8] to solve these constraints and generate new symbolic facts, as detailed in Section IV. All times are reported for a Windows 7 machine with a Xeon 64-bit 4-core CPU at 3.07 GHz with 6 GB of RAM.

We evaluate our tool on a set of 25 JavaScript applications obtained from the Windows 8 application store. For privacy and anonymization reasons, we do not mention names of individual applications, however, `http://bit.ly/Trvlux` contains some 175 sample JavaScript similar applications for Windows 8. To provide a sense of scale, Figure 9 shows line numbers and sizes of the abstract domains for these applications. It is important to note the disparity in application size compared to library stub size presented in Figure 2. In fact, the average application has 1,587 lines of code compared to almost 30,000 lines of library stubs, with similar discrepancies in terms of the number of allocation sites, variables, etc. Partial analysis takes these sizable stubs into account.

B. Call Graph Resolution

We start by examining call graph resolution. As a baseline measurement we use the standard pointer analysis provided with stubs *without* use analysis. Figure 13 shows a histogram of resolved call sites for baseline and partial inference across our 25 applications. We see that the resolution for baseline is often poor, with many applications having less than 70% of call sites resolved. For partial inference, the situation is much improved with most applications having over 70% call sites resolved. This conclusively demonstrates that the unification approach is effective in recovering previously missing flow. Full inference, as expected, has 100% of call sites resolved. Full inference, as mentioned previously, is different from partial inference in that the analysis assumes that no stubs are provided. That is, there are abstract objects that the analysis does not know about, so if unification of a symbolic object fails, full analysis assumes that this is because the object is actually some concrete object from the library which it does not know about. This implies that function call resolution for full inference succeeds in 100% of cases by construction. Note that the leftmost two bars in Figure 13 for partial inference are outliers, corresponding to a single application each.

C. Case study: WinRT API Resolution

We have applied analysis techniques described in this paper to the task of resolving calls to WinRT API in Windows 8 JavaScript applications. WinRT functions serve as an interface to system calls within the Windows 8 OS. Consequently, this is a key analysis to perform, as it can be used to compute an application's actual (as compared to declared) capabilities. This information can identify applications that are over-privileged and which might therefore be vulnerable to injection attacks or otherwise provide a basis for authoring malware. The analysis

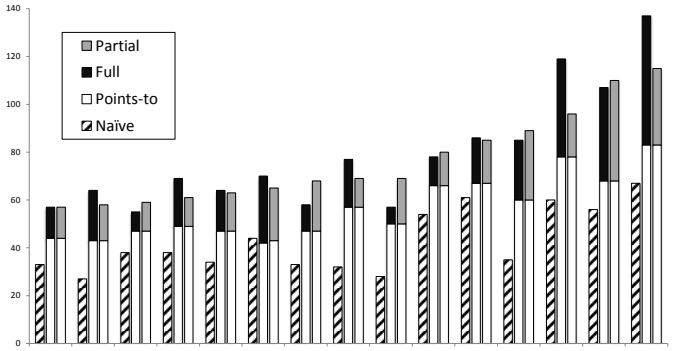


Fig. 14: Analysis techniques compared for resolving WinRT API calls.

can also be used for triaging applications for further validation and testing.

Figures in the text show aggregate statistics across all benchmarks, whereas Figure 14 represents the results of our analysis across 15 applications, those that are largest in our set. To provide a point of comparison, we implemented a naïve `grep`-like analysis by means of a JavaScript language parser which attempts to detect API use merely by extracting fully-qualified identifier names used for method calls, property loads, etc.

Techniques compared: As expected, we observe that the naïve analysis is very incomplete in terms of identifying WinRT usage. The base points-to analysis provides a noticeable improvement in results but is still very incomplete as compared to the full and partial techniques. Partial and full analysis are generally comparable in terms of recall, with the following differences:

Technique	APIs used
naïve analysis	684
points-to	800
points-to + partial	1,304
points-to + full	1,320
Total	4,108

- *All* analysis approaches are superior to naïve analysis.
- The number of API uses found by partial and full is roughly comparable.
- Results appearing only in full analysis often indicate missing information in the stubs.
- Partial analysis is effective at generating good results given fairly minimal observed in-application use when coupled with accurate stubs.
- As observed previously, common property names lead to analysis imprecision for partial analysis. Examples of common property names observed in the WinRT APIs are `name` and `version`.

Examples: We want to highlight several examples that come from us further examining analysis results.

- Observing the following call

```
driveUtil.uploadFilesAsync(
    server.imagesFolderId).
        then( function (results) {...} ))
```

  leads the analysis to correctly map `then` to `WinJS.Promise.prototype.then`.
- Observing a load of the form

```
var json = Windows.Storage.ApplicationData.current.
    localSettings.values[key];
```

  correctly resolves `localSettings` to an instance of `Windows.Storage.ApplicationDataContainer`.
- Partial analysis is able to match results without many observed uses. For instance, the call `x.getFolderAsync('backgrounds')` is correctly resolved to `getFolderAsync` on object `Windows.Storage.StorageFolder.prototype`.

1) Unification Precision: We evaluate the unification technique based on its *effectiveness* and its *precision*. We determine its effectiveness by measuring how often the technique is able to unify symbolic locations to abstract locations. Unification may fail for several reasons: (1) if no use information is available, (2) if no objects exists with all the required properties, (3) if the required properties are special properties which are explicitly excluded, such as the `length` and `prototype` properties. (4) if the symbolic or abstract object has dynamically constructed properties.

We measure precision by examining the new flow introduced by unification. Ideally, as little flow as necessary should be introduced. Introducing excessive flow will cause spurious information in the analysis. The balancing act is to be as effective as possible, without hurting precision.

Figure 10 shows how often the analysis is able to unify a symbolic location with some number of abstract locations. The figure shows numbers for both prototype and regular unification. For prototype unification the median is 8% and for regular unification it is 28%. The reason regular unification succeeds more often than prototype unification is likely because many application objects are not part of the prototype hierarchy (i.e. they are simply object literals) or there are no properties defined on the actual prototype object. We find the overall unification median to be 35%. Included in these numbers are symbolic locations for which the use analysis has no information and so cannot perform any unification. This happens, for instance, if the return value of a library function is unused.

Figure 11 shows how often the analysis is able to unify a symbolic object with a *single* abstract object. The figure shows numbers for both prototype and regular unification. For prototype unification the median is 43.8% and for regular unification it is 20.0%.

Property Names: The analysis relies on the names of properties to perform unification of symbolic and abstract objects. If the application and/or library share property names across multiple classes and objects spurious unification may occur. This does not affect soundness, but may hurt precision and scalability. The problem is typically exacerbated for applications which have been subject to minification (i.e. the process of systematically compressing the application by renaming identifiers). In our data set, we found few minified Windows 8 applications with the result that property names were sufficiently distinct for the analysis to be tractable.

Prototype Hierarchy: The prototype hierarchy of both the application and the library stubs is used to increase precision of unification. If no prototype hierarchy is available, either due to

	Category	Code	Eclipse ✓	Eclipse #	IntelliJ ✓	IntelliJ #	VS 2010 ✓	VS 2010 #	VS 2012 ✓	VS 2012 #
PARTIAL INFERENCE										

1	**DOM Loop**	`var c = document.getElementById("canvas");`
		`var ctx = c.getContext("2d");`
		`var h = c.height;`
		`var w = c.w␣`
			✗	0	✓	35	✗	26	✓	1

2	**Callback**	`var p = {firstName : "John", lastName : "Doe"};`
		`function compare(p1, p2) {`
		`var c = p1.firstName < p2.firstName;`
		`if(c != 0) return c;`
		`return p1.last␣`
		`}`
			✗	0	✓	9	✗	7	✓⋆	k

3	**Local Storage**	`var p1 = {firstName : "John", lastName : "Doe"};`
		`localStorage.putItem("person", p1);`
		`var p2 = localStorage.getItem("person");`
		`document.writeln("Mr." + p2.lastName+`
		`"," + p2.f␣);`
			✗	0	✓	50+	✗	7	✗	7

FULL INFERENCE										

4	**Namespace**	`WinJS.Namespace.define("Game.Audio",`
		`play : function() {}, volume : function() {}`
		`);`
		`Game.Audio.volume(50);`
		`Game.Audio.p␣`
			✗	0	✓	50+	✗	1	✓⋆	k

5	**Paths**	`var d = new Windows.UI.Popups.MessageDialog();` `var m = new Windows.UI.␣`	✗	0	✗	250+	✗	7	✓⋆	k

Fig. 15: Auto-complete comparison. ⋆ means that inference uses all identifiers in the program. "␣" marks the auto-complete point, the point where the developer presses Ctrl+Space or a similar key stroke to trigger auto-completion.

missing stubs or for libraries which have a flat hierarchy, less precision can be regained with this strategy. As a consequence more spurious unification may take place. In practice, both the DOM API and Windows APIs do not lead to a great deal of spurious unification due to the uniqueness of identifier names.

D. Case Study: Auto-complete

We show how our technique improves auto-completion by comparing it to four popular JavaScript IDEs: Eclipse Indigo SR2 for JavaScript developers, IntelliJ IDEA 11, Visual Studio 2010, and Visual Studio 2012. Figure 15 shows five small JavaScript programs designed to demonstrate the power of our analysis. The symbol "␣" indicates the placement of the cursor when the user asks for auto-completion suggestions. For each IDE, we show whether it gives the correct suggestion (✓) and how many suggestions it presents (#); these tests have been designed to have a single correct completion.

We illustrate the benefits of both partial and full inference by considering two scenarios. For snippets 1–3, stubs for the HTML DOM and Browser APIs are available, so we use partial inference. For Windows 8 application snippets 4–5, no stubs are available, so we use full inference. For all snippets, our technique is able to find the right suggestion *without* giving any spurious suggestions. We further believe our analysis to be incrementalizable, because of its iterative nature, allowing for fast, incremental auto-complete suggestion updates.

1) A canvas element is retrieved from the DOM. Its `getContext` and `height` properties are accessed and the programmer wishes to auto-complete the `width` property. Eclipse and Visual Studio 2010 fail to provide the right suggestion. IntelliJ IDEA provides the right suggestion amid many spurious suggestions. Visual Studio 2012 provides the right suggestion without spurious suggestions.
2) A comparator function, for person objects, is passed into a library. The `firstName` property of the first and second argument are accessed, and the programmer wishes to auto-complete the `lastName` property. IntelliJ IDEA and Visual Studio 2012 provide the right suggestion, but also includes many spurious suggestion. In fact, Visual Studio 2012 resorts to suggesting all identifiers in the file.
3) A person object is persisted to the local storage and retrieved again. The `lastName` property of the retrieved object is accessed, and the programmer wishes to auto-complete `firstName`. Only IntelliJ IDEA provides the right suggestion, among many spurious suggestions.
4) The WinJS namespace mechanism is used to populate the `Game.Audio` namespace of the global object. The `volume()` function is invoked and the programmer wishes to auto-complete the `play()` function. Again IntelliJ IDEA and Visual Studio 2012 find the right suggestion, but among many spurious suggestions.
5) The Windows namespace `Windows.UI.Popups` is accessed to create a dialog box, and then the programmer wishes to auto-complete the path `Windows.UI.␣`. Here Visual Studio 2012 is the only IDE to provide the right suggestion, but only by falling back to suggesting all identifiers in the file. IntelliJ IDEA fails to provide the right suggestion, but does provide over 250 spurious suggestions.

Several observations about IDE behavior: Eclipse never provides any suggestions when there is no flow. IntelliJ IDEA is less conservative and often suggests all possible property names (from both the application and the stubs). Visual Studio 2010 takes a middle ground and suggests properties available on the `Object⟦Proto⟧` object. Visual Studio 2012 departs from this approach and instead suggests all identifiers in the current file. To us, none of these approaches seem

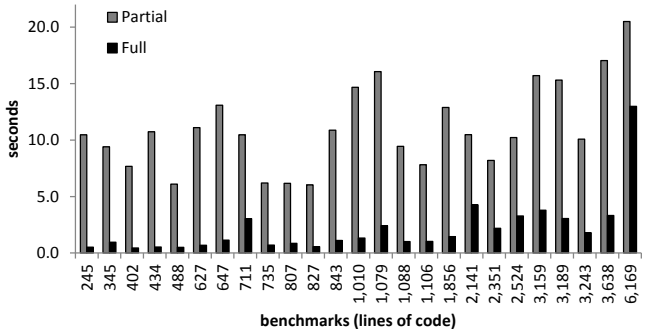


Fig. 16: Running times, in seconds. Gray is partial inference. Black is full inference. Sorted by lines of code.

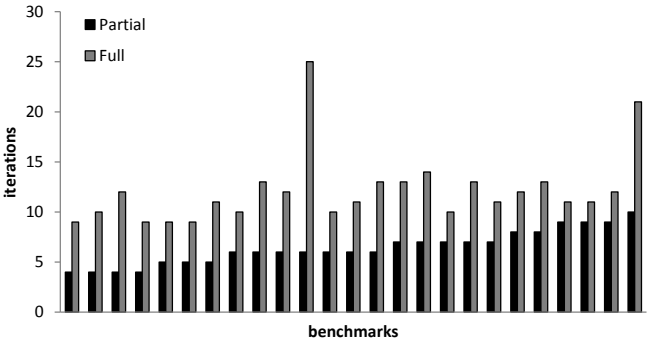


Fig. 17: Number of iterations to complete the analysis. Black is partial inference; gray is full inference. The data is sorted by the number of iterations for partial inference.

satisfactory compared to partial or full inference.

E. Analysis Running Time

Figure 16 shows the running times for partial and full inference. Both full and partial analysis running times are quite modest, with full usually finishing under 2–3 seconds on large applications. This is largely due to the fast Z3 Datalog engine.

Figure 17 shows the number iterations required to reach the fixpoint. Full inference requires approximately two to three times as many iterations as partial inference. This happens because the full inference algorithm has to discover the namespaces starting from the global object, whereas for partial inference namespaces are known from the stubs. While we do not provide a formal proof, in practice we observe that the number of iterations for full analysis is bounded by the depth of namespace nesting in our libraries.

Perhaps surprisingly, despite the extra iterations, full inference is approximately two to four times faster than partial inference. The primary reason is that the size and processing costs of library stubs are high relative to those of application code. To be fair, as applications increase in size, we would expect the speed advantage to diminish.

Figure 12 illustrates how the full inference technique discovers flow on a single application. Recall that symbolic locations are only introduced at resolved call sites and for loads with resolved base objects. In contrast there is not a similar requirement for arguments. Thus, all symbolic locations for arguments are always introduced in the first iteration whereas symbolic locations for loads and return values are introduced incrementally as more flow is discovered. The figure shows that as more and more flow is covered by the analysis less and less flow needs to be introduced until finally the fixpoint is reached.

VI. Related Work

Due to space limitations, we only provide a brief summary of related work.

Pointer analysis and call graph construction: Declarative points-to analysis explored in this paper has long been subject of research [5, 19, 27]. In this paper our focus is on call graph inference through points-to, which generally leads to more accurate results, compared to more traditional type-based techniques [11, 12, 22]. Ali *et al.* [1] examine the problem of application-only call graph construction for the Java language. Their work relies on the *separate compilation assumption* which allows them to reason soundly about application code without analyzing library code, except for inspecting library types. While the spirit of their work is similar to ours, the separate compilation assumption does not apply to JavaScript, resulting in substantial differences between our techniques.

Static analysis of JavaScript: A project by Chugh *et al.* focuses on staged analysis of JavaScript and finding information flow violations in client-side code [7]. Chugh *et al.* focus on information flow properties such as reading document cookies and URL redirects. A valuable feature of that work is its support for dynamically loaded and generated JavaScript in the context of what is generally thought of as whole-program analysis. The Gatekeeper project [13] proposes a points-to analysis based on bddbddb together with a range of queries for security and reliability. Gulfstream [14] is a successor of the Gatekeeper project whose focus is on incremental analysis and dynamic code loading. Sridharan *et al.* [24] presents a technique for tracking correlations between dynamically computed property names in JavaScript programs. Their technique allows them to reason precisely about properties that are copied from one object to another as is often the case in libraries such as jQuery. Their technique only applies to libraries written in JavaScript, so stubs for the DOM and Windows APIs are still needed.

Type systems for JavaScript: Researchers have noticed that a more useful type system in JavaScript could prevent errors or safety violations. Since JavaScript does not have a rich type system to begin with, the work here is devising a correct type system for JavaScript and then building on the proposed type system. Soft typing [6] might be one of the more logical first steps in a type system for JavaScript. Much like dynamic rewriters insert code that must be executed to ensure safety, soft typing must insert runtime checks to ensure type safety. Several project focus on type systems for JavaScript [3, 4, 25]. These projects focus on a subset of JavaScript and provide sound type systems and semantics for their restricted subsets. As far as we can tell, none of these approaches have been applied to large bodies of code. In contrast, we use pointer analysis for reasoning about (large) JavaScript programs. The

Type Analysis for JavaScript (TAJS) project [18] implements a data flow analysis that is object-sensitive and uses the recency abstraction. The authors extend the analysis with a model of the HTML DOM and browser APIs, including a complete model of the HTML elements and event handlers [17].

VII. Conclusions

This paper presents an approach that combines traditional pointer analysis and a novel use analysis to analyze large and complex JavaScript applications. We experimentally evaluate our techniques based on a suite of 25 Windows 8 JavaScript applications, averaging 1,587 lines of code, in combination with about 30,000 lines of stubs each. The median percentage of resolved calls sites goes from 71.5% to 81.5% with partial inference, to 100% with full inference. Full analysis generally completes in less than 4 seconds and partial in less than 10 seconds. We demonstrated that our analysis is immediately effective in two practical settings in the context of analyzing Windows 8 applications: both full and partial find about twice as many WinRT API calls compared to a naïve pattern-based analysis; in our auto-completion case study we out-perform four major widely-used JavaScript IDEs in terms of the quality of auto-complete suggestions.

References

[1] K. Ali and O. Lhotak. Application-only call graph construction. In *Proceedings of the European Conference on Object-Oriented Programming*, 2012.

[2] L. O. Andersen. Program analysis and specialization for the C programming language. Technical report, University of Copenhagen, 1994.

[3] C. Anderson and P. Giannini. Type checking for JavaScript. In *In WOOD 04, volume WOOD of ENTCS. Elsevier, 2004. http://www.binarylord.com/work/js0wood.pdf*, 2004.

[4] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *In Proceedings of the European Conference on Object-Oriented Programming*, pages 429–452, July 2005.

[5] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, pages 243–262, 2009.

[6] R. Cartwright and M. Fagan. Soft typing. *SIGPLAN Notices*, 39(4):412–428, 2004.

[7] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2009.

[8] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[9] P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for JavaScript. In *POPL*, 2012.

[10] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *NDSS*, 2012.

[11] D. Grove and C. Chambers. A framework for call graph construction algorithms. *Transactions of Programming Language Systems*, 23(6), 2001.

[12] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 108–124, Oct. 1997.

[13] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[14] S. Guarnieri and B. Livshits. Gulfstream: Incremental static analysis for streaming JavaScript applications. In *Proceedings of the USENIX Conference on Web Application Development*, June 2010.

[15] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy*, May 2011.

[16] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2012.

[17] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the Symposium on the Foundations of Software Engineering*, September 2011.

[18] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proceedings of the International Static Analysis Symposium*, volume 5673, August 2009.

[19] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, pages 271–286, Aug. 2005.

[20] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for java. In *LNCS 3780*, Nov. 2005.

[21] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08*, volume 5356 of *LNCS*, pages 307–325, 2008. See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

[22] A. Milanova, A. Rountev, and B. G. Ryder. Precise and efficient call graph construction for programs with function pointers. *Journal of Automated Software Engineering*, 2004.

[23] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do – a large-scale study of the use of eval in JavaScript applications. In *ECOOP*, pages 52–78, 2011.

[24] M. Sridharan, J. Dolby, S. Chandra, M. Schaefer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming*, 2012.

[25] P. Thiemann. Towards a type system for analyzing JavaScript programs. 2005.

[26] P. Thiemann. A type safe DOM API. In *DBPL*, pages 169–183, 2005.

[27] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog and binary decision diagrams for program analysis. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, Nov. 2005.