Systematic Approaches for Increasing Soundness and Precision of Static Analyzers

Esben Sparre Andreasen
Aarhus University, Denmark
esbena@cs.au.dk

Anders Møller
Aarhus University, Denmark
amoeller@cs.au.dk

Benjamin Barslev Nielsen
Aarhus University, Denmark
barslev@cs.au.dk

Abstract

Building static analyzers for modern programming languages is difficult. Often soundness is a requirement, perhaps with some well-defined exceptions, and precision must be adequate for producing useful results on realistic input programs. Formally proving such properties of a complex static analysis implementation is rarely an option in practice, which raises the challenge of how to identify causes and importance of soundness and precision problems.

Through a series of examples, we present our experience with semi-automated methods based on delta debugging and dynamic analysis for increasing soundness and precision of a static analyzer for JavaScript. The individual methods are well known, but to our knowledge rarely used systematically and in combination.

CCS Concepts **• Theory of computation → Program analysis**

Keywords Static Analysis, Soundness, Testing, JavaScript

1. Introduction

Analysis soundness Static analysis of programs written in mainstream programming languages inevitably involves approximation. Analysis designers often strive toward soundness, meaning that the analysis should consider every possible execution of the program being analyzed. Practically all analyzers deliberately treat some language features unsoundly, for example regarding reflection or native code [11]. However, unsoundness may also be caused by errors in the design or the implementation of the analysis. Such errors are easily overlooked—the analysis may produce a result, quickly and with good precision, but nevertheless a wrong result. How can the developer of a static analyzer detect such errors?

One way to ensure soundness is to make a formal proof. Sometimes this is done for key parts of the analysis design, but rarely for the entire analysis, and even more rarely for the actual implementation. One notable exception is the Verasco analyzer [8], which has been specified and proven sound using Coq. Despite the relative simplicity of that analyzer, the proof burden was massive, and the approach is hardly feasible for static analysis development in general.

Instead of requiring analyzers to be *provably* sound, we aim for making them *probably* sound, which can be achieved using thorough, automated testing. One such approach is property-based testing (i.e., quickchecking), which has been shown in previous work [12] to be an effective technique for detecting errors in static analyzers, by exploiting the generic algebraic properties of lattices and dataflow constraints. In this paper, we describe our experience with another pragmatic technique that we call *soundness testing*. The idea is simple and unsurprising: after a program has been analyzed statically, we compare the analysis results with the concrete states that are observed by a dynamic analysis of the program. If the information produced by the static analysis does not over-approximate the information obtained from the executions, a soundness error has been detected.

Analysis precision Another important aspect of static analysis design is precision. As approximation is inevitable and higher precision generally implies higher worst-case complexity, the right choice of abstractions can only be determined experimentally. Analysis precision is usually measured using some analysis client, for example the ability to prove absence of certain kinds of errors in the programs being analyzed. However, internal analysis metrics, such as sizes of points-to sets or degrees of suspiciousness of abstract values [1] may also provide valuable hints to where it may be advantageous to improve the analysis abstractions. In this paper we focus on another technique to investigate analysis precision problems, which is inspired by the work on *blended analysis* [4, 16]. A blended analysis is a static analysis that uses observations from a dynamic analysis to unsoundly approximate the program behavior. Tuned static analysis [10] is a related technique that uses unsound static pre-analysis instead of dynamic analysis. The previous work on blended and tuned analysis is about specific analysis algorithms that increase precision (by sacrificing soundness), whereas our goal here is to systematically identify opportunities for improving an analysis design.

Although our approach is inspired by blended analysis, it is also related to the recently proposed process called *root-cause localization and remediation* [17] for supporting the design of JavaScript analyses. That process involves a static analysis that automatically identifies where it looses precision, and a mechanism for suggesting alternative context sensitivities for those locations based on dynamic analysis.

Delta debugging In addition to the use of soundness testing and blended analysis, soundness problems and precision problems are both amenable to *delta debugging* [18]. This is an effective technique to reduce the size of inputs (e.g. programs to be analyzed) while preserving problematic behavior, which in our case is unsoundness or low precision.

Contributions In this paper we briefly describe our experience with soundness testing and blended analysis as methods for increasing soundness and precision of the TAJS [1, 7] static analyzer for JavaScript. Both of these techniques rely on information recorded by the same dynamic analysis. We have used both soundness testing and blended analysis in combination with delta debugging to identify causes and importance of soundness and precision problems. The methods are semi-automated and tightly integrated into the TAJS infrastructure, and they have become essential tools for guiding our continuous development of the analyzer.

TAJS We believe the techniques we present are broadly applicable to static analysis development in general, but we here focus on the TAJS analyzer.[1] In brief, TAJS is a whole-program dataflow analyzer for JavaScript, aiming to infer type-related properties involving the flow of primitive values, objects, and functions in the programs being analyzed. It supports most ECMAScript 5 features, including the native library and large parts of modern browser API and HTML DOM functionality.

Regarding soundness, we face several challenges. The language itself is extremely complex, there is a substantial native library specified in the ECMAScript standard, and the browser API and HTML DOM are not only massive but also poorly documented and constantly evolving. Tiny errors in the models can easily cause serious soundness issues that may affect validity of experimental results if not detected.

Regarding precision, we (and many others) have found that some programming patterns that are common in widely used JavaScript libraries require extraordinary analysis precision, and that inadequate precision often renders even small programs unanalyzable due to avalanches of spurious dataflow [1, 9, 10, 14, 17]. Here, "unanalyzable" means, for example, that the analysis (spuriously) finds that `eval` is called with an unknown string as argument.

2. Basic Techniques

In this section we briefly describe the three basic techniques with examples of how we have used them in our ongoing development of TAJS.

[1] TAJS is available at `http://www.brics.dk/TAJS/`.

```
1  var a, b, x;
2  a = {p: 0, q: 0};
3  b = [];
4  for (var p in a)
5   b.push(p);
6  x = b[0]
7  a[x] = b[x];
8  a.p();
```

(a) Reduced program for 5 versions of underscore.js.

```
1  var a, x;
2  a = {};
3  a.p = 0;
4  b.q = 0;
5  for (var i = 0; i++) {
6   x = Object.keys(a)[i];
7   this[x] = a[x];
8  }
```

(b) Reduced program for 11 versions of lodash.js.

Figure 1: Delta debugging precision problems.

2.1 Delta Debugging

Delta debugging [18] is a technique for automated debugging of programs. The essence of the technique is that a large input is reduced systematically while preserving some specific buggy behavior. The output is valuable because it makes it easier to understand *why* the buggy behavior arises. A delta debugging session takes two inputs: (1) an input program to reduce, and (2) a predicate that determines if the (reduced) program exhibits some specific behavior. The output is a smaller program exhibiting this behavior. We will use the term "delta debugging" throughout this text, although a more appropriate name is the generalized concept of "cause reduction" [6]. The difference is that we are using the technique to identify causes of a wide range of analysis behaviors, and not just buggy behaviors.

Example While further developing our static determinacy analysis technique [1], we observed that several utility libraries[2], each consisting of thousands lines of code, were unanalyzable. To investigate whether the libraries contained common patterns that were problematic for TAJS to analyze, we applied delta debugging using the predicate that TAJS should not be able to analyze the library within 5 minutes. Manually inspecting the outputs that were produced for the different versions of the libraries quickly revealed a small, common pattern for each library. Figure 1a shows the resulting reduced program that was common to all five different versions of the underscore.js library. The manageable size (only 8 lines) made it possible to determine the root cause of the critical precision loss: The entries of an array are mixed together since the iteration order of `for-in` loops is implementation-specific according to the ECMAScript standard. The lost precision eventually causes spurious dataflow to a large number of native functions at the method call in line 8. A similar reduced program for a common pattern in 11 versions of the lodash.js library can be seen in Figure 1b.

Delta debugging in practice The JavaScript delta debugger JSDelta[3] has been used by several JavaScript research groups. JSDelta systematically simplifies a JavaScript program by performing statement deletion, sub-expression simplification, and general purpose program optimizations, such as function inlining.

[2] `http://underscorejs.org/`, `https://lodash.com/`, among others

[3] `https://github.com/wala/jsdelta`

JSDelta is integrated with TAJS through a simple Java interface. To start delta debugging, a predicate is implemented in 5–10 lines of Java code, and the delta debugging main method is executed through the IDE with the predicate and some input program. This often reduces a few thousand lines of code to less than 20 within a few hours, fully automatically. The integration through Java also enables highly specialized predicates. As an example, a specialized predicate has been used to understand surprising differences between two slightly different analysis configurations. This predicate was defined to determine whether one configuration would lead to a particular kind of flow graph node being processed significantly more often than with the other configuration.

Although a delta debugger in principle can produce output that contains problems that are not present in the input, we have found that situation to be rare. In practice, the output usually exhibits the problem that was also present in the input, which makes the approach useful for understanding analysis limitations.

2.2 Soundness Testing

We use the term *soundness testing* to denote the process of checking whether observations in a concrete execution of a program are subsumed by the results computed by the static analysis. Soundness testing has been applied in various ways to many static analyzers. A notable example of this is a study of the consequences of *deliberate* unsoundness in the Clousot analyzer [3]. An interesting conclusion in that work is that Clousot often encounters unsoundness in practice but nevertheless rarely misses alarms.

Value logging An important design choice when performing soundness testing is deciding what information to include from the dynamic executions. We have chosen a simple approach based on *value logs*, which consist of the values of expressions that are computed during the execution of a program. Other options include recording the call graph [17], statement traces [16], or state snapshots [5, 13]. We have found the simpler value logs sufficient for our purposes.

An example program and (a simplified version of) its value log produced by our tool can be seen in Figure 2. The property access on line 2 of the program is represented by the first two lines of the value log. It has been logged that the property access occurred on the object allocated at position `1:8` in the program (`BASE`), and that the result is the string "`foo,bar`" (`PROP`). Similarly, the call to `split` on line 3, is represented by the three last lines of the value log.

A notable design choice for our value logs is that we abstract away from execution order. This allows us to eliminate duplicate entries, which leads to a considerable reduction of the sizes of the logs.

Another important choice is that the value logs do not contain information about call stacks or scope chains of function objects. As mentioned, TAJS is partly context-sensitive, but it is extremely difficult to implement a faithful mapping from, for example, runtime call stacks to the abstract notion of contexts used by the static analysis. This means that when checking subsumption of the concrete values and the abstract values, we can only report a soundness error if a given concrete value is not subsumed by the corresponding abstract values for *all* contexts.

```
1 var o = { p: 'foo,bar' };
2 var s = o.p;
3 var a = s.split(',');
```

```
f.js:2:9 BASE OBJECT(f.js:1:8)
f.js:2:9 PROP STRING("foo,bar")
f.js:3:9 BASE STRING("foo,bar")
f.js:3:9 CALLEE BUILTIN(String.prototype.split)
f.js:3:9 ARGO STRING(",")
```

Figure 2: A program (top) and its value log (bottom).

Value logging in practice We obtain value logs with a dynamic analysis implemented using Jalangi [15]. The program of interest is instrumented and executed such that observations about runtime values are recorded. When the execution ends, the observations are post-processed into a value log, which is then persisted for reuse. The value log also contains metadata, for example a checksum of the program code so that we can easily detect if the program has been modified and the value log should be recreated.

The logging mechanism also supports different environments, enabling the creation of value logs for plain ECMAScript applications, Node.js applications, and browser-based applications. For example, if a log file is missing for a browser-based application, a browser is spawned to load the instrumented application, making it easy to manually interact with it and decide when to stop recording.

Example As an example of a failing soundness test, consider the code and the value log in Figure 2. If the analysis is missing a model for the `split` property of string objects, then our soundness testing tool fails with the following report:

```
Soundness testing failed for 1/5 checks:
 - CALLEE on program line 3:
  - concrete: BUILTIN(String.prototype.split)
  - abstract: {undefined}
```

In this case, it is easy for the analysis designer to spot and fix the root cause of the unsoundness. All that is needed is a model of the built-in function `String.prototype.split`.

Soundness errors can easily spread in less obvious ways: a missing assignment to a field can cause the soundness check of the subsequent reads of that field to fail because of an unsound value rather than missing dataflow. Such extraneous soundness errors can make it harder to deduce the root cause of unsoundness. Furthermore, there may be multiple root causes of a failing soundness test, which can also make it harder to identify a single one of them. In Section 3.1 we present a technique for remedying these situations.

Soundness testing in practice Soundness testing is integrated into TAJS' regression test system and has been successful in uncovering many subtle soundness bugs. Initially, we found bugs in the core parts of the analysis, but recently mostly in the models of the huge, complex, and constantly evolving native libraries. For this reason we are planning to apply deeper checks of values originating from the native libraries, for example, not just checking that a value expected to be an object (rather than a primitive value) really is an object, but also that the object has the right properties.

Soundness errors sometimes result in highly inaccurate analysis results, which may be difficult to notice without soundness testing. However, as also observed by Christakis et al. [3], unsoundness can be benign, in the sense that it sometimes influences only a few nearby statements and not the remainder of the program, nor the analysis output.

In TAJS, we maintain a catalog of known soundness errors, which are then ignored when running the soundness tests. Most of these known errors have been added to the catalog because they have been classified as benign. This catalog helps to document the unsoundness in TAJS, as advocated by the soundiness manifesto [11]. It also helps prioritizing which soundness errors to fix.

At the time of writing, the main regression test suite of TAJS contains approximately 2 200 successfully soundness tested JavaScript programs, comprising 900 000 individual soundness checks for 100 000 syntactic locations. Only around 100 of the soundness checks fail, due to around 20 different soundness bugs that are caused by, for example, inadequate modeling of the HTML DOM.

2.3 Blended Analysis

An easy way to increase the precision of a static analysis is to replace parts of the abstract states with concrete states obtained by a dynamic analysis. While this is obviously not sound in general, it is sound relative to the execution path taken by the dynamic analysis.

The idea is not new. Blended analysis [4, 16] for Java and JavaScript allows the analysis to follow the control flow of the concrete execution. The TamiFlex tool [2] uses the same approach to handle Java reflection. Dynamic determinacy analysis [14] for JavaScript is based on a similar idea but retains soundness by only using dynamic information that is valid for all executions.

We apply this technique by leveraging the value logs that we already have from soundness testing as described in Section 2.2. When analyzing a program, the associated value log can be queried for the concrete values at a program location. The abstract value for that program location can then be refined by *intersecting* (technically, applying the greatest lower bound) with the abstraction of the concrete values. Another option would be to *replace* the abstract value with the abstraction of the concrete values, but since the value logs do not record any control flow information, that would generally be less precise.

In practice, our value logs are more detailed than presented in Section 2.2. We record some relational information, for example, at a property write, we log the base object, the property value, and the value to be written. Thereby, when refining, for example, the abstract value being written, we can ignore concrete values that apply to other abstract objects and other property values, which increases precision.

```
var message = x == y ? "Same" : "Different";
var code = "print('" + message + "')";
eval(code);
```

```
f.js:3:1 ARGO STRING("print('Same')")
```

Figure 3: A program with `eval` and a line from its value log.

Example As a TamiFlex-like example, a static analysis for JavaScript can use blended analysis for the argument to `eval`.[4] Consider the program and its associated value log in Figure 3. Without having support for determining that the variables `x` and `y` always have the same value, the analysis is able to evaluate the `eval` call as the code `print('Same')`.

A similar use of blended analysis that enables focused prototype analysis design is to obtain call and points-to graphs from the value log instead of approximating them soundly.

Blended analysis in practice The use of blended analysis makes it possible to circumvent challenging language or library features, allowing the analysis designer to proceed with other aspects of the analysis.

We mostly use blended analysis in combination with other techniques, as we describe in Sections 3.2 to 3.4. However, we have also used the approach to investigate "best-case scenarios" for analyzing large JavaScript applications that are beyond reach of all existing sound JavaScript analyzers. By applying blended analysis aggressively—at *all* program locations—we can test the analyzer for fundamental scalability problems and logical implementation errors. Any errors that are detected in such a scenario also exist without enabling blended analysis but may be more difficult to find without it.

3. Combining the Basic Techniques

The basic techniques introduced in the previous section can be combined to create some particularly powerful techniques that guide the design of static analyses.

3.1 Soundness Testing and Delta Debugging

As stated in Section 2.2, soundness testing can expose soundness bugs, but it is often difficult to locate the cause of a failing soundness test if the program being analyzed is large or if the bug causes many soundness checks to fail. Delta debugging is extremely useful in these cases. Each iteration of this delta debugging process works as follows. (1) run the program concretely to obtain a value log, (2) analyze the program, (3) perform soundness testing of the result from step 2 using the value log from step 1. The delta debugging predicate is that step 3 results in one or more failing soundness checks. Delta debugging then automatically produces a small program containing a soundness error.

[4] We note that TAJS is able to handle some common occurrences of `eval`.

```
1  var i, s;
2  i = "0";
3  s = i++;
```

```
Soundness testing failed:
 - VAR 's' on program line 3:
  - concrete: NUMBER(0)
  - abstract: {STRING("0")}
```

Figure 4: Reduced program with a subtle soundness error.

If one wants to target a specific soundness error, then the predicate can be refined to consider only that particular error. Nevertheless, any reduced program with a soundness error is valuable even after the error has been fixed, since their small sizes make them useful as fast regression tests.

Example Figure 4 shows a reduced version of an unsoundly analyzed program, together with the failing soundness test. This small program was produced starting from a large program that at that time had thousands of soundness failures. The reduced program exposed that the value of a postfix expression in JavaScript is, perhaps surprisingly, the number-coerced value and not the original value. In this example, it turned out that the exposed soundness bug was benign, and after fixing it the original program still had thousands of soundness failures. However, repeating the process quickly revealed that those failures all had the same root cause and could also easily be fixed.

3.2 Soundness Testing and Blended Analysis

By combining soundness testing and blended analysis, it is possible to detect soundness errors even in programs that are unanalyzable (in the sense described in Section 1) when using the ordinary analysis! Using the same dynamic information for the two purposes, any failures that are detected during the soundness testing must be due to unsoundness in the analysis, and *not* due to the under-approximation introduced by the use of blended analysis.

Example Consider the soundness error below:

```
Soundness testing failed for 43/3932 checks:
 - PROP on program line 542:
  - concrete: BUILTIN(Symbol.unscopables)
  - abstract: {undefined}
```

It reveals that the program being analyzed uses the `Symbol` ECMAScript 6 feature, which was not yet fully modeled in TAJS at the time this test was run. Without the use of blended analysis, the program was unanalyzable due to inadequate analysis precision, and it would have been difficult to tell that the feature was not just encountered due to spurious dataflow.

3.3 Delta Debugging and Blended Analysis

We can also combine delta debugging and blended analysis. This time, delta debugging is not instantiated with a program, but instead with a set of program locations where blended analysis is allowed. This combination of techniques gives a way of finding a minimal set of locations that need to be handled precisely by the static analysis.

Delta debugging is initiated with the set of all locations in the program to be analyzed and the predicate that TAJS can analyze the program, for example within one minute, by applying blended analysis in the current set of locations. The outcome is a reduced set of locations where blended analysis is critical. Manually inspecting those locations often gives good hints for improving the analysis design.

We find that the resulting number of locations is usually below 5, which supports the claim that few root causes of imprecision can render the analysis result useless [17].

Examples Using this technique to investigate causes of precision problems when analyzing various small applications of the underscore.js library resulted in the following automatically generated report.

```
underscore-1.8.3.js needs more precision at:
- PROPERTY WRITE at line 3
```

The relevant piece of code is shown in Figure 5 (line numbers have been modified to match the figure). Blended analysis was only needed in a single location, which means that improving TAJS to be able to handle this particular location precisely was the key to analyze the entire program. TAJS' problem with underscore.js was that the abstract value of `name` was an unknown string, so each property of `obj` was conservatively written to each property of the library object, thereby introducing a critical loss of precision.

```
1  _.mixin = function(obj) {
2   _.each(_.functions(obj), function(name) {
3    var func = _[name] = obj[name];
4    _.prototype[name] = function() {
5     func.apply(_, arguments);
6    };});};
7  _.mixin(_);
```

Figure 5: Excerpt from problematic underscore.js code.

Further in our investigation involving a more complicated application of the library, we got this report:

```
underscore-1.8.3.js needs more precision at:
- PROPERTY WRITE at line 3
- CALL at line 5
```

This time, one additional location needed more precision. The problem was that `func` could be every element in `obj`, which would cause TAJS to conservativly model calls to every function of `obj`, making the program unanalyzable.

Using this technique to systematically investigate precision problems with a range of applications of the library, we obtained an overview of the various precision bottlenecks, which was useful for prioritizing our effort and designing useful improvements of the analysis abstractions.

3.4 Combining All Three Techniques

As discussed in Section 3.2, combining blended analysis and soundness testing makes it possible to detect soundness errors even with programs that cannot be analyzed by TAJS. It is not always easy to identify the root cause of such a soundness error, but again we can make use of delta debugging to automatically produce a small program that is analyzed

```
1 function f(){
2  return arguments;
3 }
4 f().p;
```

```
Soundness testing failed:
- PROP on program line 4:
  - concrete: UNDEFINED
  - abstract: {}
```

Figure 6: Identifying the cause of unsoundness from an unanalyzable program.

unsoundly. Compared to the combination of soundness testing and delta debugging described in Section 3.1, we now use the value log that is created in each delta debugging step both for soundness testing and for blended analysis.

Example As mentioned in Section 3.3, TAJS could not analyze simple applications of the underscore.js library, meaning that without blended analysis, we could not use those JavaScript programs to detect soundness errors. By combining the three techniques we not only detected a soundness error, but we also obtained a reduced program containing the error. The reduced program and the soundness test report are shown in Figure 6. It turned out that the analysis did not properly support accessing properties of the special `arguments` object outside of its declaring function. The reduced program made it easy to locate the cause. The single fix reduced thousands of failing soundness checks to zero, since the soundness error influenced the dataflow in the rest of the program.

4. Conclusion

We have presented our experience with soundness testing, blended analysis, and delta debugging for systematically guiding improvements of soundness and precision of the TAJS static analyzer. Both soundness testing and blended analysis build on top of value logs obtained by dynamic analysis. Other useful techniques, such as quickchecking and suspiciousness metrics, are described elsewhere [1, 12],

Our experience can be summarized as the following recommendations to static analysis developers:

- Use dynamic analysis to record value logs for all benchmark programs. The value logs are useful for improving both soundness and precision as the analysis design and implementation evolves.
- Use soundness testing as an integrated part of the development, and maintain a catalog of known soundness issues. When soundness errors appear, use delta debugging to quickly identify the cause.
- When precision problems appear, use blended analysis to investigate how alternative analysis abstractions may help. Combining blended analysis with delta debugging can often automatically locate the critical places where extra precision is needed.
- By combining soundness testing, blended analysis, and delta debugging, it is possible to quickly identify soundness errors even for programs that are unanalyzable due to insufficient analysis precision.

Acknowledgments

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

References

[1] Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. In *OOPSLA*.

[2] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*.

[3] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2015. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *VMCAI*.

[4] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended analysis for performance understanding of framework-based applications. In *ISSTA*.

[5] Asger Feldthaus and Anders Møller. 2014. Checking correctness of TypeScript interfaces for JavaScript libraries. In *OOPSLA*.

[6] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2016. Cause reduction: delta debugging, even without bugs. *STVR* (2016).

[7] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *SAS*.

[8] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A formally-verified C static analyzer. In *POPL*.

[9] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *ESEC/FSE*.

[10] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. 2015. Practically tunable static analysis framework for large-scale JavaScript applications. In *ASE*.

[11] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundness: a manifesto. *Commun. ACM* (2015).

[12] Jan Midtgaard and Anders Møller. 2015. Quickchecking static analysis properties. In *ICST*.

[13] Joonyoung Park, Inho Lim, and Sukyoung Ryu. 2016. Battles with false positives in static analysis of JavaScript web applications in the wild. In *ICSE SEIP*.

[14] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic determinacy analysis. In *PLDI*.

[15] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE*.

[16] Shiyi Wei and Barbara G. Ryder. 2013. Practical blended taint analysis for JavaScript. In *ISSTA*.

[17] Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. 2016. Revamping JavaScript static analysis via localization and remediation of root causes of imprecision. In *ESEC/FSE*.

[18] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *STE* (2002).