Improving Precision of JavaScript Program Analysis with an Extended Domain of Intervals

Astrid YOUNANG, Lunjin LU
Computer Science and Engineering Dept.
Oakland University
Rochester Hills, Michigan
{awaindja, l2lu}@oakland.edu

***Abstract*—Abstract Interpretation has been a promising approach for static analysis of JavaScript programs. Static analysis is used for security auditing, debugging, optimization and error checking. JavaScript is dynamically typed, uses prototype-based inheritance and first class functions. It supports reflective calls, access to object fields and allows object fields to be dynamically added and deleted. These dynamic features make JavaScript flexible to use. At the same time, they make JavaScript applications more susceptible to programming errors. The challenge that comes with the analysis of such programs is the design of abstract domains that will precisely track properties of interest without affecting performance. This paper presents our work on improving analysis precision of JavaScript programs. We used an extended domain of intervals to track ranges of numeric values of variables. This is the first time interval domain has been applied to the analysis of the full JavaScript language. We implemented the new abstract domain within a JavaScript abstract interpreter. Our experiments show that the new abstract domain enables the abstract interpreter to infer more precise type information for most of the benchmark programs and strikes a good balance between analysis precision and cost. While the analysis of some benchmarks take more time as expected, some other benchmarks actually take less time.**

***Index Terms*—JavaScript, static analysis, abstract interpretation, numeric abstract domain, interval analysis**

I. Introduction

JavaScript is widely used for web applications. Web browsers allow JavaScript to interact with users, react to their actions and modify the content of webpages. Because of that, JavaScript code manipulates data from users which might be very sensitive (e.g. passwords) or potentially unsafe and harmful. A lot of optimization and security analysis tools for JavaScript web applications have been proposed during the past few years. They are based on static analysis [1] [2] [3], dynamic analysis [4] [5], hybrid analysis [6] [7] [8], security enforcement [9], etc.

Static analysis, which is the automatic discovery of progam properties, has been used for compiler optimization, program verification and program understanding. Many static analysis tools are based on abstract interpretation. Abstract interpretation is a general theory of semantic approximations and has been formalized by Patrick and Radhia Cousot in [10] and [11]. A program is analyzed in an abstract domain of program properties equipped with transfer functions. A transfer function describes the effect of a program instruction on its input program properties. An abstract domain is often a complete lattice. The correctness of the analysis is formalized by a Galois connection between the concrete and the abstract domains [12]. The precision and cost of the analysis depend on the abstract domain.

During the past few years, several abstract interpretation-based static analyzers have been proposed for JavaScript programs. Logozzo et al. in [13] designed and implemented Rapid Atomic Type Analysis (RATA in short). They combined an interval analysis, kind analysis and variation analysis to detect variables that always take 32 bit integers as their values. This information is used in a Just-In-Time compiler to allocate these variables to registers and generate specialized instructions. This analysis is designed for JIT compiler optimization and ignores many JavaScript constructs such as object creation, closures and throw expressions [13]. Jensen et al. in [3] presented TAJS - Type Analysis for JavaScript. TAJS tracks undefinedness, nullness, type and point-to information and uses recency abstraction to increase analysis precision. TAJS detects definite type errors in JavaScript programs and generate warnings on potential type errors. Vineeth et al. in [2] designed and implemented JSAI - JavaScript Abstract Interpreter, an abstract interpreter for static analysis of JavaScript programs. The framework is designed to make context sensitivity an independent concern and can be configured for different context sensitivities. Lee et al. proposed SAFE [14], a Scalable Analysis Framework for EcmaScript. It provides three kinds of intermediate representations for JavaScript that can be used in various analyses and optimizations. SAFE detects range, reference, syntax, type and URI errors and successfully generates warnings such as the reading of an absent property of an object or a conditional expression that is always true or false. TAJS, JSAI and SAFE use constant propagation domains for numbers to track numeric information, which is much simpler than the interval domain in RATA, or the interval domain we are using.

Interval domain for program analysis was first introduced by Cousot in 1976 [15] and it has been well studied. Recently, it has been applied to JavaScript program analysis [13].

In this work, we present an extended abstract domain of intervals. The main contributions of this paper are as follows. Firstly, we designed and implemented an extended domain of

intervals and abstract operators for all JavaScript operations. We augmented the classic interval domain with new abstract elements and applied it to the full JavaScript language. The domain keeps track of numeric information more precisely than the numeric abstract domains used in TAJS, JSAI, SAFE and RATA. Secondly, we performed an empirical performance and precision evaluation on benchmarks. We used Google V8 and Sunspider benchmarks [16] [17] and also the browser addons, and machine generated programs taken from JSAI [2]. In most of the benchmarks, the precision has been increased with an acceptable increase of the analysis time. Some benchmarks actually take less time because the increased precision on the numeric domain helps avoid unnecessary computation in the overall analysis.

The rest of the paper is organized as follows. Section II presents a motivating example and section III the extended domain of intervals and abstract operations over this domain. Section IV presents the evaluation of the new analysis in terms of precision and performance and section V discusses related work on abstract interpretation-based static analyses of JavaScript programs and interval analysis. We conclude and discuss future work in section VI.

II. Motivating example

We will use the following JavaScript code to illustrate the impact of the extended abstract domain of intervals on the precision and performance. The signs "%" and "$" indicate the results obtained by the extended domain of intervals and the original numeric domain used in JSAI respectively.

```
1  var i=-120;          %i=-120‘          $i=-120
2  var j=0;             %j=0              $j=0
3  var k=0;             %k=0              $k=0
4
5  while(i<-5){
6    j=i;               %j=Int(-120,-6)   $j=NReal
7    i=i+1;             %i=Int(-120,-6)   $i=NReal
8  }
9
10 if(j<9){             %True             $Bool.Top
11   while(k<400){
12     print("");
13     k=k+1;           %k=Int(0,399)     $k=NReal
14   }
15 }
16 else{
17   print("");
18 }
```

Analysis with JSAI's original numeric domain discovers that *i*, *j* and *k* variables are unsigned 32-bit integers. This is indicated by the abstract element *NReal*. It fails to recognize that i is negative at the end of the execution. The truth value of the condition $(j < 9)$ in the line 10 cannot be precisely determined using this information. This is because $j < 9$ is true for some unsigned integer values for j and it is false for some other values. The analysis indicate this using the abstract value element Bool.Top. So, both the *then* and the *else* branches need to be analyzed and their analysis results need to be merged using the join operator. With the extended abstract domain of intervals, the precision is increased as we can better approximate value ranges for the variables: $i \in Int(-120, -5)$, $j \in Int(-120, -6)$ at the time the *if* statement is to be analyzed. The condition $(j < 9)$ in line 10 is evaluated to be true and the *else* branch is skipped. In JSAI, a primitive value is described as a tuple of abstract values that capture numeric, boolean, location, nullness and undefinedness [2]. The abstract semantics in JSAI uses the reduced product [11] of these component abstract domains. Such a configuration enables abstract domains to take each other's information to improve their precision. In the example, the precision gained from the extended domain of intervals makes analysis of the condition in line 10 more precise. The extended domain of intervals will help detect infeasible paths during the analysis of the program and will decrease the number of false type error warnings reported by the analysis. Loop condition expressions are also used to constrain the possible values of variables, which also improves analysis precision.

III. Abstract Domain of Intervals

In this section, we describe the extended abstract domain of intervals, the operations over these intervals, the widening and narrowing operators that are used to ensure termination of analysis.

*1) **Extended Intervals**:* All JavaScript numbers are 64-bit floating point numbers as defined by the IEEE754 standard [18]. The range of numbers is from (-5e-324) to (1.79769e+308). There are also three special numbers - *NaN*, *PositiveInf* and *NegativeInf*. The domain of intervals for program analysis was first introduced by Cousot in [15]. This same abstract domain was used to demonstrate that, by carefully using widening and narrowing operators, we can gain in precision and efficiency than using a finite abstract domain [19]. It was later extended by Logozzo in RATA - an online static analysis tool for Just-In-Time compiler optimization for JavaScript programs [13]. The abstract domain used in our work is obtained by extending RATA's interval domain.

The extended abstract domain of intervals is $INT_{JS} = \{\top, \bot, NaN, Int32\} \cup \{NConst(a) \mid a \in \mathbb{R}\} \cup \{Int(a,b), Norm(a,b) \mid a,b \in \mathbb{Z} \cup \{-\infty, +\infty\}, a \leq b\}$. *Norm(a,b)* describes the set of real numbers between *a* and *b* including *NaN*, *Int(a,b)* the same set of real numbers without *NaN*. *NaN* is the special number arising from computations such as $\infty \times 0$ or $0 \div 0$. *NConst(c)* describes the real number *c*, and *Int32* the set of all unsigned 32-bit integers. The abstract domain is redundant since two abstract elements may be equivalent to each other in that they describe the same set of concrete elements. For instance, *Int(2,2)* and *NConst(2)* both describe $\{2\}$. A reduced domain can be constructed by merging equivalent abstract elements into equivalence classes and using equivalence classes as abstract elements [20]. We forgo such a construction and simply use an arbitrary element of an equivalence class to represent the equivalence class and make sure that each abstract operation produces equivalent outputs from equivalent inputs.

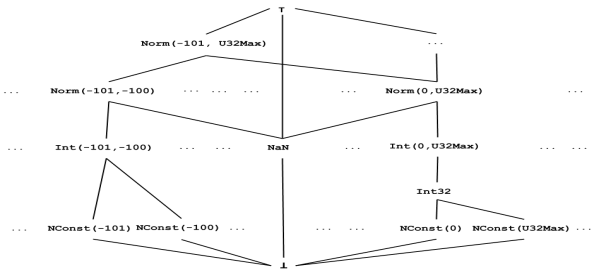

Fig. 1: Illustration of the Extended Domain of Intervals

Figure 1 illustrates the lattice of intervals and the order relation $\sqsubseteq$ between intervals. U32Max is the maximal integer of 32 bits - 4294967295. The domain of intervals has infinite increasing chains. Let $\sqcup$ and $\sqcap$ denote the least upper bound and the greatest lower bounds operators respectively. Note that *Int(0,U32Max)* and *Int32* are two distinct abstract elements. *Int32* is the set of unsigned 32-bit integers and is mostly used with bitwise shift operators. Let Val be the set of concrete JavaScript values.

$$Val \quad = \quad \mathbb{R} \cup \{-\infty, +\infty\} \cup \{NaN\} \cup String \cup Object$$

where $Object$ is the set of JavaScript Objects and $String$ is the set of all string constants. The concretization function $\gamma : INT_{JS} \longrightarrow \mathbb{P}(Val)$ for the extended abstract domain of intervals is defined as

$$\begin{aligned}
\gamma(\bot) &= \{\} \\
\gamma(NaN) &= \{NaN\} \\
\gamma(Norm(a,b)) &= \{r | r \in \mathbb{R}, a \leq r \leq b\} \cup \{NaN\} \\
\gamma(Int(a,b)) &= \{r | r \in \mathbb{R}, a \leq r \leq b\} \\
\gamma(Int32) &= \{n | n \in \mathbb{N}, 0 \leq r \leq 2^{32}-1\} \\
\gamma(NConst(a)) &= \{a\} \ for \ a \in \mathbb{R} \\
\gamma(\top) &= Val
\end{aligned}$$

Consider an abstract state in which $x \in Norm(2,3)$. The evaluation of $x < 4$ in this abstract state will result in Bool.Top because *NaN* is in $\gamma(Norm(2,3))$ and $NaN < 4$ is false. In order to avoid loss of precision in such cases, we introduced a new abstract interval $Int(a,b)$ such that $\gamma(Int(a,b))$ does not contain NaN. The evaluation of $x < 4$ in an abstract state in which $x \in Int(2,3)$ will result in True. Our lattice structure is different from Logozzo's in that all Logozzo's intervals contain the concrete element *NaN*. Instead, we clearly separate intervals containing *NaN- Norm(·,·)* from intervals without *NaN- Int(·,·)*. In addition, we have abstract elements *Int32* and *NConst(·)*. *Int32* is captured in RATA by a separate piece of information. These design decisions had a huge impact on the precision, especially those of boolean operations.

The abstract operator approximating a concrete operator $\odot$ is denoted $\hat{\odot}$. The abstract operators on the extended domain of intervals include $\hat{+}, \hat{-}, \hat{\times}, \hat{\div}, \hat{\equiv}, \hat{mod}, \hat{\ll}, \hat{\gg}, \hat{\ggg}, \hat{<}, \hat{\leq}$, $Bitwise\ A\hat{N}D$, $\hat{OR}$, $X\hat{O}R$ and $N\hat{O}T$ that are approximations of corresponding concrete operators defined by the EcmaScript specification 5 [21]. In addition, we have a widening operator ∇ and a narrowing operator Δ. Let functions $f_0, f_1 : INT_{JS} \longrightarrow \mathbb{B}$ be defined such that $f_0(I)$ is true if and only if $0 \in \gamma(I)$ and $f_1(I)$ is true if and only if $\gamma(I)$ contains ∞. The functions f_0 and f_1 are used by multiplicative operators $\hat{\div}$, $\hat{\times}$ and $\hat{mod}$ to check whether or not an interval contains 0 or ∞. Their definitions via simple case analysis on the structure of the input are omitted.

Figure 2 defines three special abstract operations on intervals of the forms $Norm(\cdot,\cdot)$ and $Int(\cdot,\cdot)$. The functions f_0 and f_1 are used to detect computations that may produce the value NaN such as in $\infty \times 0$, $0 \div 0$, $\infty \div \infty$, $\infty \ mod \ a$, $a \ mod \ 0$ with $a \in \mathbb{R}$. Accordingly, the functions *min* and *max* were revised to ignore *NaN* when minimal and maximal elements are computed.

*2) **Condition expressions in loops** :* Condition expressions are used to constrain the abstract state and refine the values of variables. For example, a variable x with abstract value $Int(0,100)$ entering a loop with the condition $x > 20$ has its abstract value updated to $Int(20,100)$ before entering the loop. Condition expressions can be simple such as $x == y$, $x > y$ or composite such as $((2x - y > z)$ && $(k + a! = 2y))$. In simple expressions, it is relatively easy to calculate the value of the variables, however, it is not an easy job to obtain the resulting values of variables involved in composite conditional expressions. Special functions are used to update the values of the variables involved in conditional expressions. More details can be found in [22] and [23].

IV. Evaluation

The new abstract domain can be used with any abstract interpretation-based analysis framework without major modifications. We implemented it on top of JSAI and replaced JSAI's original numeric domain. JSAI is a framework written in Scala. We ran JSAI on a Scientific Linux 6.3 distribution with 24 Intel Xeon CPUs with a capacity of 1.6GHz and 32GB memory. The benchmarks chosen are the standard SunSpider [17] and V8 programs [16], 7 browser addons programs from the Mozilla addon repository [24], 7 machine generated JavaScript code from the Emscripten LLVM test suite [25] and 3 real world JavaScript programs taken from *codeplex.com* and *defensivejs.com*, which are open source JavaScript frameworks.

The precision metric used in our benchmark programs is the number of program locations that may generate a type error warning. Type errors occur in situations such as when the program calls a non-function value as a function and tries to update, read or delete a property of an object that is null or undefined. We analyzed all the programs with the original numeric domain, and with the extended abstract domain of intervals in order to compare results issued from the same platform. The results are promising in most of the benchmarks. Figure 3 represent the percentage decrease of

$$
\begin{array}{rcl}
N_1 \hat{\times} N_2 & = & Norm(floor(min(x)), ceil(max(x))) \\
I_1 \hat{\times} I_2 & = & \begin{cases} Norm(floor(min(x)), ceil(max(x))) & \text{if } ((f_0(I_1) \wedge f_1(I_2)) \vee (f_1(I_1) \wedge f_0(I_2))) \\ Int(floor(min(x)), ceil(max(x))s) & \text{otherwise} \end{cases} \\
N_1 \hat{\div} N_2 & = & Norm(floor(min(y)), ceil(max(y))) \\
I_1 \hat{\div} I_2 & = & \begin{cases} Norm(floor(min(y)), ceil(max(y))) & \text{if } ((f_0(I_1) \wedge f_0(I_2)) \vee (f_1(I_1) \wedge f_1(I_2))) \\ Int(floor(min(y)), ceil(max(y))) & \text{otherwise} \end{cases} \\
N_1 \ \hat{mod} \ N_2 & = & \begin{cases} Norm(a, 0) & \text{if } b \leq 0 \\ Norm(0, b) & \text{if } a \geq 0 \\ N_1 & \text{if } f_0(N_1) \end{cases} \\
I_1 \ \hat{mod} \ I_2 & = & \begin{cases} Int(a, 0) & \text{if } b \leq 0 \wedge \neg f_1(I_1) \wedge \neg f_0(I_2) \\ Int(0, b) & \text{if } a \geq 0 \wedge \neg f_1(I_1) \wedge \neg f_0(I_2) \\ N_1 & \text{if } f_1(I_1) \vee f_0(I_2) \\ I_1 & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 2: Abstract operations $\hat{\times}, \hat{\div}$ and $\hat{mod}$ where $I_1 = Int(a, b)$, $I_2 = Int(c, d)$, $N_1 = Norm(a, b)$, $N_2 = Norm(c, d)$, $x = \{ac, ad, bc, db\}$ and $y = \{\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\}$.

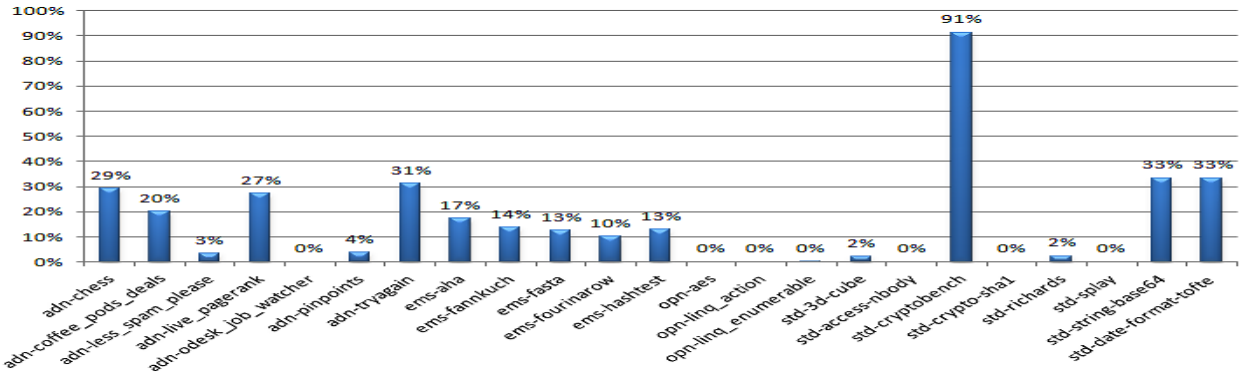

Fig. 3: Decrease of type errors warnings. The higher the percentage, the better.

program locations that may generate a type error warning. Both analyses are based on over-approximation and generate a certain number of false type error warnings. The decrease in the number of type error warnings is an indication of a gain in precision. The average percentage of the decrease is 14.95%. The minimum decrease is 0% and the maximum is 91% in the *std-cryptobench.js* file. A range error can be encountered when trying to create an array of an illegal length. The number of range errors warnings reported by the analysis has also been reduced. In the *std-cryptobench.js* file, it went from 38 to 4 and in *crypto-aes.js*, from 2 to 0. The extended abstract domain of intervals infers more precise information than the original numeric domain of JSAI. Hence, the increase of the analysis time is expected. In addition, the analysis time is increased by the widening and narrowing operations. The comparison is shown in Figure 4 between the execution time taken by the original JSAI and the modified one. We analyzed each program 10 times and computed the average of execution time. On most of the benchmarks, the results obtained in the figure meet the expectations. However, as explained in the motivating example, the increase of precision can have a good turnaround on the performance. The use of the reduced product to improve precision enables the number, boolean, string, nullness and undefinedness components of an abstract base value to benefit from each other. Therefore, the precision of the new interval domain improves the precision of the boolean domain and can avoid unnecessary computation in branches of conditional statements. That is the reason why JSAI with the extended abstract domain of intervals takes less time than the original JSAI on some benchmarks.

A. *Infeasible paths and error reporting*

Static analysis derives interesting properties of programs before they are actually executed. Over approximation causes a lot of infeasible paths to be taken, thus a certain number

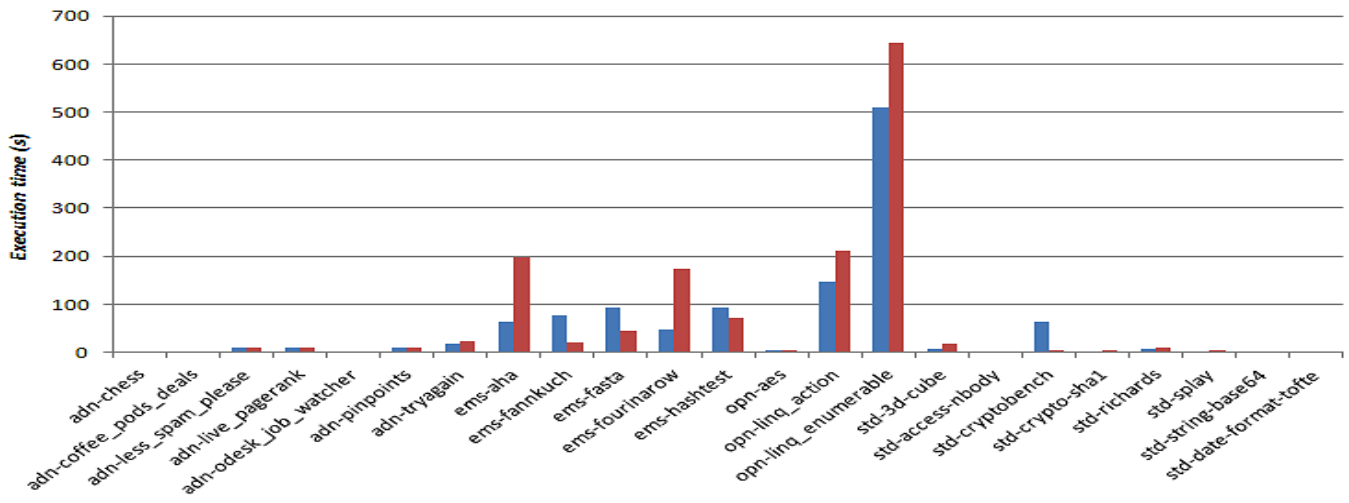

Fig. 4: Comparison on analysis time. In red, the new analysis which takes more time as expected because the new number domain is more precise. In blue is the execution time in *s* of the original analysis.

of false positive errors to be added in the analysis report. It is important to decrease the number of false positive errors reported by the analysis because checking them manually is a tedious and error-prone job. Since interval analysis is able to detect some infeasible paths, the analyzer will not go trough those paths and will not report additional false positive errors. This is measured by the total number of states processed by the worklist algorithm. Table 1 illustrates benchmark programs for which the decrease in the number of states with the extended domain of intervals is over 40%. In addition to that, we can better infer the value ranges of variables, which can be used for further analysis.

TABLE I: Benchmark programs in which the number of states processed by the worklist algorithm has decreased over 40%

	Total # of states	
Benchmarks	with infeasible paths	without infeasible paths
ems-fannkuch	5371	2811
ems-fasta	5450	2890
ems-hashtest	5484	2924
std-cryptobench	3645	728

V. Related Work

In this section, we focus on major JavaScript analysis frameworks and their numeric abstract domains.

JavaScript has mainly been analyzed for security and optimization purposes. Sound and unsound approaches have been proposed in [26], [27], [9], [28] and [29] to detect security vulnerabilities in browser extensions and JavaScript web applications. Due to the typeless nature of JavaScript, type inference analysis has received a lot of attention from [30], [31], [32], [33], [3], [34], [35] and [13]. Contributions have also been made on pointer analysis in [36], which can be used for further JavaScript analyses.

SAFE in [14] is a static analysis framework for Ecmascript. Based on abstract interpretation, it provides three kinds of intermediate representations for JavaScript. SAFE provides an analyzer capable of detecting type related errors in JavaScript programs. SAFE uses the constant propagation domain for numbers extended with additional elements, which is similar to TAJS' numeric domain and therefore less precise than our abstract numeric domain.

Jensen et al. in [3] presented TAJS, a type analysis tool for JavaScript based on abstract interpretation. TAJS makes use of a novel lattice structure to provide sound and detailed type information in JavaScript programs. TAJS has been evaluated on Google V8 and Sunspider benchmarks and has successfully detected type errors in call/construct sites, variable read instructions and property access read/write/delete instructions. TAJS has evolved and improved its precision over the years by using techniques such as lazy propagation and recency abstraction. Their abstract numeric domain is the constant propagation domain for numbers extended with three additional elements - *INF*, *UInt* and *NotUInt*. *INF* is the least upper bound of $-\infty$ and $+\infty$, *UInt* that of all unsigned integers and *NotUInt* that of all other float-point numbers. The abstract numeric domain of TAJS is less precise than ours, for example, the least upper bound of the numbers 1 and 3 is represented by the abstract elements *UInt* and *Int(1,3)* respectively in TAJS and our interval domain. JSAI and TAJS' approaches are similar in that they both detect type related errors in JavaSript programs. However, JSAI is more configurable than TAJS in terms of choice of sensitivity and TAJS is more mature and contains some precision and performance optimizations such as recency abstraction and lazy propagation [2]. As pointed out, the numeric domains used in TAJS, JSAI and SAFE are constant propagation domains and these domains are less precise than the interval domains used in RATA and our analysis.

Logozzo and Venter present RATA - a static analysis based on abstract interpretation that is used by a Microsoft JavaScript engine [13]. It infers type information that is used by the

JavaScript engine to generate specialized versions of functions for different type profiles. The inferred type for a local variable or parameter includes the kind of the values that the variable can take and its range. The kind information tells whether the values of the variable are definitely 32-bit integers, or definitely 64-bit floating point numbers as specified by the IEEE754 standard [18] and allows the JIT compiler to allocate variables in integer or float-point registers. The range information is kept by an extension to the abstract domain of intervals invented by Cousot and Cousot [11]. The range information is needed to infer precise kind information as operations on 32-bit integers values may yield values that are not 32-bit integers. The analysis assumes the worst-case senario for global variables. Since it is an online analysis used by a JIT compiler, more emphasis is put on analysis efficiency in RATA than in other static analyses. In RATA, the abstract numeric domain contains abstract elements *NaN*, *Normal(a,b), OpenLeft(a), OpenRight(a)*. *Normal(a,b)* is the set of real numbers between a and b with $a, b \in \mathbb{Z}$, including *NaN*. *OpenLeft(a)* and *OpenRight(a)* are the sets of real numbers that are less than a and greater than a respectively with $a \in \mathbb{Z}$, including *NaN*. All abstract elements in RATA's interval domain contain *NaN*, our interval domain has abstract elements that do not contain *NaN* - *NConst(·), Interval*$(\cdot, \cdot)$ and *Int32*, which has positive effects on analysis precision as explained before.

VI. Conclusion and Future Work

We designed an extended domain of intervals for the full JavaScript language and implemented it on top of JSAI. In most of the benchmarks, the precision has been increased with an acceptable increase of the analysis time. In some other benchmarks, the analysis time has been reduced because unnecessary computation is avoided due to the detection of infeasible paths in the program. Though we have only tested the extended domain of intervals with JSAI, it can be used with other abstract interpretation-based analysis for JavaScript. Future work would investigate other numeric abstract domains such as polyhedra, octagon and octahedron and will analyze the tradeoff between analysis precision and cost.

References

[1] S. Guarnieri et al., Saving the world wide web from vulnerable javascript, in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 177–187, ACM, 2011.

[2] V. Kashyap et al., Jsai: a static analysis platform for javascript, in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[3] S. H. Jensen, A. Møller, and P. Thiemann, Type analysis for javascript, in *Static Analysis*, pages 238–255, Springer, 2009.

[4] P. Saxena, S. Hanna, P. Poosankam, and D. Song, Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications., in *NDSS*, 2010.

[5] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, ACM SIGPLAN Notices **48**, 165 (2013).

[6] O. Tripp and O. Weisman, Hybrid analysis for javascript security assessment, in *ESEC/FSE*, volume 11, 2011.

[7] P. Saxena et al., A symbolic execution framework for javascript, in *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528, IEEE, 2010.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, Staged information flow for javascript, in *ACM Sigplan Notices*, volume 44, pages 50–62, ACM, 2009.

[9] S. Guarnieri and V. B. Livshits, Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code., in *USENIX Security Symposium*, pages 151–168, 2009.

[10] P. Cousot and R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, ACM, 1977.

[11] P. Cousot and R. Cousot, Systematic design of program analysis frameworks, in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, ACM, 1979.

[12] A. Cortesi, Widening operators for abstract interpretation, in *Software Engineering and Formal Methods, 2008. SEFM'08. Sixth IEEE International Conference on*, pages 31–40, IEEE, 2008.

[13] F. Logozzo and H. Venter, Rata: rapid atomic type analysis by abstract interpretation–application to javascript optimization, in *Compiler Construction*, pages 66–83, Springer, 2010.

[14] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, Safe: Formal specification and implementation of a scalable analysis framework for ecmascript, in *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2012.

[15] P. Cousot and R. Cousot, Static determination of dynamic properties of programs, in *Proceedings of the second International Symposium on Programming*, pages 106–130, Paris, 1976.

[16] http://v8.google.com/svn/data/benchmarks/v7/run.html.

[17] http://webkit.org/perf/sunspider/sunspider.html.

[18] D. Zuras et al., IEEE Std 754-2008 , 1 (2008).

[19] P. Cousot and R. Cousot, Comparing the galois connection and widening/narrowing approaches to abstract interpretation, in *Programming Language Implementation and Logic Programming*, pages 269–295, Springer, 1992.

[20] P. Cousot and R. Cousot, The Journal of Logic Programming **13**, 103 (1992).

[21] E. ECMAScript et al., Ecmascript language specification, 2011.

[22] A. Ermedahl and M. Sjodin, Interval analysis of c-variables using abstract interpretation, Citeseer, 1996.

[23] Y. Wang, Y. Gong, J. Chen, Q. Xiao, and Z. Yang, An application of interval analysis in software static analysis, in *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, volume 2, pages 367–372, IEEE, 2008.

[24] https://addons.mozilla.org/en-US/firefox/.

[25] http://www.emscripten.org/.

[26] A. Taly et al., Automated analysis of security-critical javascript apis, in *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 363–378, IEEE, 2011.

[27] S. Bandhakavi et al., Communications of the ACM **54**, 91 (2011).

[28] R. N. Taylor, D. L. Levine, and C. D. Kelly, Software Engineering, IEEE Transactions on **18**, 206 (1992).

[29] A. Guha, S. Krishnamurthi, and T. Jim, Using static analysis for ajax intrusion detection, in *Proceedings of the 18th international conference on World wide web*, pages 561–570, ACM, 2009.

[30] P. Thiemann, Towards a type system for analyzing javascript programs, in *Programming Languages and Systems*, pages 408–422, Springer, 2005.

[31] P. Heidegger and P. Thiemann, Recency types for analyzing scripting languages, in *ECOOP 2010–Object-Oriented Programming*, pages 200–224, Springer, 2010.

[32] R. Chugh, D. Herman, and R. Jhala, ACM SIGPLAN Notices **47**, 587 (2012).

[33] S. H. Jensen, P. A. Jonsson, and A. Møller, Remedying the eval that men do, in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 34–44, ACM, 2012.

[34] S. H. Jensen, A. Møller, and P. Thiemann, Interprocedural analysis with lazy propagation, in *Static Analysis*, pages 320–339, Springer, 2011.

[35] C. Anderson, P. Giannini, and S. Drossopoulou, Towards type inference for javascript, in *ECOOP 2005-Object-Oriented Programming*, pages 428–452, Springer, 2005.

[36] D. Jang and K.-M. Choe, Points-to analysis for javascript, in *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1930–1937, ACM, 2009.