Fixpoint Reuse for Incremental JavaScript Analysis

Lawton Nichols
University of California, Santa Barbara
Santa Barbara, USA
lawtonnichols@cs.ucsb.edu

Mehmet Emre
University of California, Santa Barbara
Santa Barbara, USA
emre@cs.ucsb.edu

Ben Hardekopf
University of California, Santa Barbara
Santa Barbara, USA
benh@cs.ucsb.edu

Abstract

Frequently updated programs cause the cost of static analysis to be multiplied by the number of program versions. When the baseline cost is high (for example, analyzing JavaScript), this multiplicative factor can be prohibitive. As an example, JavaScript-based browser addons are continually updated and there are known instances where malicious code has been injected into such updates; thus the addons must be repeatedly vetted each time an update happens.

Incremental analysis reduces this cumulative cost by reusing analysis results of previous versions to reduce the cost of analyzing an updated version. However, existing incremental analyses are not applicable to dynamic programming languages such as JavaScript because they make assumptions that don't hold in this setting. In this paper, we propose the first incremental static analysis for JavaScript. We do not require perfect precision, but we show empirically that there is negligible precision loss in practice. Our technique includes a method for matching code between JavaScript program versions, a non-trivial problem which existing techniques do not solve. For our benchmarks, drawn from real browser addons and node.js programs, our incremental analysis performance is on average within a factor of two of an optimal incremental analysis.

CCS Concepts • **Theory of computation → Program analysis**; • **Software and its engineering → Reusability**; General programming languages.

Keywords incremental program analysis, javascript analysis

ACM Reference Format:
Lawton Nichols, Mehmet Emre, and Ben Hardekopf. 2019. Fixpoint Reuse for Incremental JavaScript Analysis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3315568.3329964

SOAP '19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6720-2/19/06…$15.00
https://doi.org/10.1145/3315568.3329964

1 Introduction

JavaScript programs are an integral part of the internet ecosystem, from the server to the client, and present a tempting target for malicious actors. For example, JavaScript-based browser addons have complete access to the browser's state and can do anything they want with that information, including collecting and disseminating users' sensitive data; examples of such behavior have been found in the wild [6, 8]. Thus, JavaScript is an important target for static analyses that attempt to ensure safety and security. Numerous such analyses have been published, e.g., to ensure that browser addons do not leak sensitive information [21, 36, 37].

However, a single-time static analysis is not sufficient when programs are continually updated with new versions (as is the case with browser addons). There are known instances where malicious code has been snuck into existing JavaScript programs during such updates [3]. Thus, static analyses must be run on every version of a program, not just the first one. However, JavaScript is a highly dynamic and difficult to analyze language, and the resource cost can be high. If there is a central entity serving as the main gateway for these programs (e.g., browser addon repositories) that is responsible for running all of these analyses, they must shoulder the bulk of this cost. Being forced to re-run the analyses for every new program version only exacerbates these problems.

The contribution of this paper is a technique called *fixpoint reuse* to mitigate the performance problems attendant on repeatedly statically analyzing the same JavaScript program over multiple versions.

Our technique falls under the general rubric of *incremental static analysis*, a topic that has been extensively studied over the years. However, no existing work deals with a dynamic language such as JavaScript. In particular, the existing work generally relies on two major assumptions: (1) an *a priori* known flow-graph model of the program; and (2) a known mapping between the old and new program versions. Unfortunately, JavaScript programs do not have a simple flow-graph model and require expensive static analysis to compute precise control-flow and data-flow information. In addition, previous works assume that the mapping between

versions is given as input to the analysis (without describing how it is computed) or that the mapping is trivially computable from the known flow-graph (which JavaScript doesn't have). Thus, the existing works' assumptions do not hold and those techniques are not immediately applicable to languages such as JavaScript.

We rely on two key insights to reposition incremental static analysis for JavaScript: (1) the problem of matching between two program versions is similar to the problem of clone-detection, and thus we can leverage existing clone-detection techniques [13, 20, 33]; and (2) whereas modern incremental analyses are precise (i.e., yield the same answer as a non-incremental analysis), we can relax the requirement for precision while still getting useful results. That is, our incremental analysis can yield additional false positives beyond what a from-scratch analysis would yield, but we show empirically that this does not happen very often. Together, these insights enable our technique to achieve speedups within 2× of an *optimal incremental analysis*, which we define as an incremental analysis between identical program versions, thus allowing maximum reuse.

In the context of a central gateway such as a browser addon repository that is analyzing third-party programs, another benefit of our technique is that it does not rely on the gateway having to store past analysis results for every program that it analyzes. Analysis summaries of previous versions can safely be left to the third-party developers to store and transmit with any program updates; our technique guarantees that the results of the analysis will still be sound. The most that a malicious developer could do is to degrade the performance and precision of the incremental analysis up to some limit, after which we would fall back to a normal from-scratch analysis. Our technique is flexible enough to handle a variety of scenarios that distribute the analysis work between the central authority and the app developer in different ways, while still allowing the central authority to guarantee the soundness of the results.

Although we implemented fixpoint reuse for analysis of JavaScript, our technique is not Javascript-specific. Fixpoint reuse can be used for building incremental analyses for other dynamic languages where a priori, precise control-flow information is not available.

2 Related Work

In this section we review the work on incremental static analysis to put our technique in context.

2.1 Incremental Analysis via Restarting Iteration

Perhaps the most closely related work to our technique is from the early '80s. There are three works that present a technique called *restarting iteration* [16–18]. Unlike our fixpoint-reuse work, restarting iteration assumes a known control-flow graph and a provided mapping from old to new program version. Similarly to our fixpoint-reuse technique, the technique does not guarantee a precise incremental analysis, i.e., it could introduce additional false positives. The main contributions of our work in relation to this old work are (1) removing the assumption of a known, simple flow-graph, thus making the technique applicable to dynamic languages such as JavaScript; and (2) providing a method to compute a mapping between program versions rather than assuming one will be provided, thus making the technique more practical.

2.2 Precise Incremental Analysis

Starting in the late '80s the work on restarting iteration was abandoned in favor of techniques that guarantee precise results—i.e., analyses that return the same results as a non-incremental analysis. This flavor of incremental analysis has dominated the field since that point [12, 14, 15, 19, 22, 23, 26–30, 32, 34, 35]. Modern incremental analyses focus on pruning old results that might negatively impact precision. There have been a number of advancements, but all are for non-dynamic languages with simple flow-graph program models and assume that either the version mapping is provided or can be trivially computed from the respective flow-graphs. None of the precise incrementalization methods is immediately applicable to languages such as JavaScript.

2.3 "Incremental" Analysis of JavaScript

Livshits and Guarnieri [25] present Gulfstream for streaming JavaScript programs. The word "incremental" is used in a different context in that paper: the analysis is incremental in the sense that it statically analyzes all JavaScript code that it can, and then when dynamic processes load *new JavaScript files*, those files are analyzed in an incremental fashion. The paper presents a points-to analysis of JavaScript that is unsound and makes use of analysis result invalidation; whereas our work maintains soundness, is a general abstract interpretation, and does not invalidate any previous information.

3 Fixpoint Reuse Overview

In this section we describe the problem that we are solving and the basic ideas of our approach, called *fixpoint reuse*. We stay at a relatively high level, focusing on the central concepts.

Problem Definition. The three inputs are P_{prior} (the prior version of the program), FP_{prior} (the fixpoint analysis solution for P_{prior}), and P_{upd} (the new version of the program). We assume FP_{prior} is in the form of a map from program points (potentially including calling context information) to abstract states representing the solution at that point. Let FP_{upd} be the fixpoint solution for a from-scratch analysis of P_{upd}. Our goal is to compute $FP_{\widehat{upd}}$, an over-approximation of FP_{upd}.

Method Overview. Our approach is to (1) compute a partial mapping $P_{prior} \rightarrow P_{upd}$ from program points in P_{prior} to program points in P_{upd} that correspond with high confidence, then (2) use $P_{prior} \rightarrow P_{upd}$ to seed the initial analysis state for $FP_{\widehat{upd}}$ with the abstract states for corresponding program points as given in FP_{prior}. We then (3) analyze P_{upd} starting from the seeded initial analysis state and ensure that we visit every program point in P_{upd} at least once in order to guarantee a sound analysis: because every transfer function is monotone, and because we only ever increase the size of the inputs to the transfer functions, our analysis results are always an overapproximation of the from-scratch analysis results.

Program Matching Challenges. Computing the map $P_{prior} \rightarrow P_{upd}$ can have extreme effects on the efficacy of the incremental analysis. Besides adding or removing statements between versions, functions may have been renamed, allocation sites may have been moved, and calling sequences may have been modified. Thus matching between versions must include renaming calling contexts and abstract heap locations from FP_{prior} to the appropriate cognates in P_{upd}. When matching there are three possibilities: (1) We correctly match, (2) we incorrectly match, or (3) we cannot match.

The first case is the best case; the more correct matches we compute the more effective the incremental analysis will be in improving performance. The third case, while not ideal, isn't too harmful; the incremental analysis won't benefit from the prior analysis, but it can compute the information in the same way as a from-scratch analysis.

The second case, however, is by far the worst case and highlights the non-triviality of the matching problem. An incorrect match means that the incremental analysis will be seeded with incorrect information from the prior analysis. While this incorrect information doesn't affect the soundness of the results, it does mean that the incremental analysis must propagate this incorrect information to all reachable program points, reducing performance and precision.

Thus, it is far better to fail to match a program point than it is to incorrectly match a program point. Our matching algorithm must carefully balance between matching often and matching well. Failing to match often enough means that we get no performance improvement; failing to match well means that we get both performance and precision *reduction*. We find that techniques borrowed from clone detection work well because they provide a list of potential correspondances with a measure of how close the correspondance is, allowing us to tune the tradeoffs described above.

4 Fixpoint Reuse for SAFE

Our prototype implementation is built on top of SAFE version 2.0. The SAFE JavaScript analysis framework [24] does not perform its analysis at the level of the original JavaScript source code. Instead, the source is translated to a simpler intermediate representation (IR) that is more amenable to analysis—it breaks complicated expressions into simpler ones, and makes explicit the implicit operations of the JavaScript language (e.g., type coercion, argument array construction before a function call, etc.). SAFE IR consists of a list of functions, each of which consists of a list of basic blocks (hereafter refered to as blocks), each of which consists of a list of instructions.

In order to reuse analysis results, we must therefore create a correspondence mapping between programs at the level of SAFE's IR. In the rest of this section, we describe our implementation at a high level. A more detailed description including specific matching algorithms can be found in the accompanying tech report [31]. We focus on the program matching aspect of our technique, as once a mapping between versions has been computed the actual analysis is straightforward.

4.1 Program Matching

When we match JavaScript programs at the SAFE IR level, we match functions, blocks, and instructions, in that order. Once we are confident that two functions correspond, we then match their blocks, and once we believe we have chosen the best block correspondence we match individual instructions. Matching instructions is necessary for two main reasons: (1) correctly translating calling contexts from FP_{prior} to $FP_{\widehat{upd}}$ requires accurate mapping of call instructions, and (2) correctly translating abstract heap addresses from FP_{prior} to $FP_{\widehat{upd}}$ requires accurate mapping of allocation instructions. In both cases, failing to accurately match corresponding program entities does not cause unsoundness but results in imprecision and will cause the analysis to visit unnecessary program locations and heap addresses. Thus matching with high confidence is crucial for our method's accuracy and efficiency.

4.2 Function Matching

Our function matching algorithm is based on an edit-distance calculation. The algorithm is parameterized by a function Criteria that computes the distance between pairs of functions as a numerical score. We consider two functions to "match" when the criteria is below a certain threshold. We instantiated Criteria with different choices in order to evaluate which combination of distance criteria worked best—for functions, for example, this critera consists of comparing line numbers, function IDs, instruction counts, and identifier usage. Given the distances, the algorithm matches those functions with the best distance score that is under our empirically-calculated threshold.

Table 1. Open-Source Benchmarks. For every sequence of benchmark versions (e.g., [A, B, C]), we compare the closest pairs (i.e., (A, B) and (B, C)).

Benchmark Name	Version A	Lines	Version B	Diff	Distance
chess1 [2]	0.1.0.1	283	0.1.1.2	127+/116−	44
chess2	0.1.1.2	295	0.1.1.3	40+/10−	69
emoji-helper1 [5]	1.1.0	579	1.1.1	17+/3−	24
emoji-helper2	1.1.1	594	1.2.0	15+/1−	10
simple-translate [9]	2017.09.25	301	2017.10.14	2+/2−	0
k-cup-deals [7]	1.2	499	1.3	12+/0−	63
dateformat1 [4]	2011.03.13	166	2012.11.08	49+/7−	22
dateformat2	2012.11.08	208	2013.03.11	15+/8−	6
dateformat3	2013.03.11	216	2014.11.28	201+/55−	44
dateformat4	2014.11.28	261	2017.09.18	11+/6−	10
yallist1 [11]	2015.12.19	585	2017.03.11	24+/16−	8
yallist2	2017.03.11	594	2017.03.13	9+/0−	8
yallist3	2017.03.13	602	2017.04.25	2+/0−	0
balanced-match [1]	0.4.2	193	1.0.0	93+/102−	161
url-join1 [10]	2.0.0	149	2.0.1	1+/1−	0
url-join2	2.0.1	149	2.0.2	1+/1−	0

4.3 Block and Instruction Matching

Blocks are also matched using a similar edit distance calculation. Instructions are matched based on the type of the instruction, the number of allocation sites appearing in the instruction, and the names of the variables involved (modulo generated numerical suffixes).

5 Evaluation

In this section we evaluate the efficacy of fixpoint reuse in terms of performance and precision. Because we are guaranteeing the soundness of the incremental analysis, we must at a minimum visit every program point in the updated version at least once. Thus, the potential for speedup lies in reducing the number of times the analysis has to revisit a program point before convergence. The quality of the program matching between versions will play a large role.

We want to study the efficacy of fixpoint reuse on actual programs from the wild. We take four JavaScript-based browser addons and four Node.js programs along with between 1–4 updates for each program taken from available public repositories. These benchmarks are described in Table 1. We are limited in our benchmark selection by SAFE's capabilities—these benchmarks were chosen from a set of smaller programs because SAFE can completely model their code and analyze them using a reasonable amount of resources. Following previous work on analyzing browser addons [21], we edit the original code to provide stubs for built-in browser functions, and we include some amount of driver code to ensure that the analysis visits all interesting locations in the source file. We manually selected sources and sinks for each file.

The actual analysis that we perform on these benchmarks is a taint analysis implemented using the SAFE JavaScript analysis infrastructure, suitably modified to implement fixpoint reuse. The implementation is available online.[1] We use the taint results to measure the precision of the incremental analysis versus a from-scratch analysis.

Table 2. Handmade Benchmarks: Richards (v8 lines of code: 546)

Versions (A–B)	Diff	Distance
v0−v8	30+/2−	64
v1−v8	28+/2−	60
v2−v8	23+/2−	51
v3−v8	21+/2−	43
v4−v8	19+/1−	38
v5−v8	13+/0−	33
v6−v8	11+/0−	27
v7−v8	8+/0−	16

To help calibrate expectations, we start with a limits study to determine the maximum speedup the incremental analysis could possibly get. We accomplish this by running the incremental analysis on "updated" benchmark versions that are exactly the same as the original, thus ensuring a perfect program match and minimal revisiting of program points. The results are in Section 5.1.

Another factor that comes into play is how different the original and updated programs are. In the extreme, the updated program could be completely different from the original and not benefit from incremental analysis at all. To help understand the effect of program "distance", we have created a set of handmade benchmarks and a series of successively more "distant" updates for each benchmark, allowing us to study the effects of program distance in a controlled manner. The results are in Section 5.2.

Finally, we compare the speedups that we achieve on the actual updated program versions to determine how close to the optimal results we are.

5.1 Limits Study

For our limits study we take each program version of each benchmark and run an incremental analysis on itself—in other words, we take the from-scratch analysis and apply fixpoint reuse to exactly the same program. This is the ideal case for reuse and provides the maximum benefit. Because we have a perfect program match, the only cost in the incremental case is for visiting each program point exactly once. We run three different experiments varying context-sensitivity from 0-CFA to 2-CFA; a "program point" for a context-sensitive analysis includes the context. The results are not shown for space reasons, but the speedups observed were from 1.12× to 14.46×.

[1] Our implementation is located at http://www.cs.ucsb.edu/~pllab under the "Downloads" link.

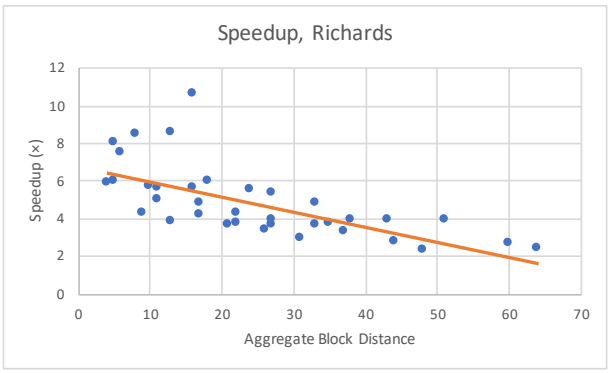

Figure 1. Handmade Results: Richards

5.2 Controlled Distance Study

Table 2 contains information on one of our handmade benchmarks: it is the v8 Richards benchmark with statements deleted. We also tested with the v8 Navier-Stokes benchmark (not shown for lack of space); the results are similar. We made random (but attempted to avoid program-breaking) deletions—these files are then "played backwards" to appear as a sequence of code additions. Thus, successive versions contain greater and greater differences to the original version.

Our distance metric is derived from our program matching algorithm. Given two programs A and B, we compute the set of matching function pairs and, for each pair, we compute the block edit distance. The sum of the block edit distances over all matching function pairs is our measure of distance between A and B. We investigated several other possible distance metrics and found that they all behaved similarly.

We chose this methodology because additions seem to be the most common updates to code: in our real-world, open-source benchmarks, each commit contains over 4× the number of additions to deletions on average. Of the four outliers, only one was a legitimate case of refactoring; others were superficial changes regarding whitespace or test suite configuration, and so these diffs were exaggerating the truth.

Figure 1 shows the results of Richards handmade benchmark. We ran every combination of version pairs that respected the order, e.g., v1~v2, v1~v3, v1~v4, v2~v3, v2~v4, v3~v4, etc. We grouped each pair of programs based on their distance score. These 1CFA analysis results paint a picture of how program additions impact reuse.

The Richards benchmark consists of several small functions—for the original benchmark, the fixpoint took 9,081 iterations to converge, there were 494 unique program points visited, and there were 3 loops. The chosen taints were calculated precisely for all version pairs.

5.3 Real-World Evaluation

We run the real-world version updates at three context-sensitivity levels. The time it takes to perform the program matching process on a given version pair is the same for every context sensitivity level; they are all quite small (under 5s, with the majority around 1s). For the longer-running analyses this number is negligible.

5.3.1 Incremental Results

We compare our method to a baseline of running the static analysis on the updated version of each benchmark from scratch (i.e., with fixpoint reuse turned off).

Table 3 shows the results of reuse for each different context sensitivity level. We find that all taints are carried over with very little imprecision. The dateformat benchmark is the only case with imprecise taints, and this is due to the modeling of a JavaScript built-in object that causes the analysis to return the $\top_{\text{addr}}$ address (i.e., the abstract address corresponding to all concrete addresses). Because the heap is prepopulated with extra information, there are more locations to point to than in the from-scratch case.

All in all, while maintaining soundness and high precision in a proof-of-concept taint analysis, our fixpoint reuse method allows us to more than double the speed of an analysis on average for real-world programs.

For another perspective, Table 4 shows our speedup relative to our best possible incremental analysis results (i.e., the observed speedup divided by the optimal speedup). These results provide another means of observing program difference: the version pairs with the fewest differences have either an optimal or close-to-optimal speedup. On average, our reuse method is within a factor of two of the optimal speedup for these benchmarks, and we believe this is representative of the general case.

6 Conclusion and Future Work

We have presented fixpoint reuse, a method for incremental program analysis that reuses fixpoint analysis solutions from one version of a program to accelerate the analysis of an updated version of the same program by matching program points between the two versions. We have applied fixpoint reuse to JavaScript analysis and shown that we get good results on real-world JavaScript programs.

The maximum performance improvement that our technique can provide is limited by the necessity to visit every program point at least once during the incremental analysis. As future work, it would be interesting to investigate techniques to skip visiting program points that are unlikely to have changed while still guaranteeing a high probability of soundness; doing so could potentially increase the performance improvement of incremental analysis even further.

Acknowledgments

This work was supported by NSF grant CCF-1319060.

Table 3. Results relative to from-scratch analysis. Tainted sink state counts in black were the same as the baseline counts, while counts in red exhibited imprecision and had one extra tainted sink.

Benchmark	0CFA		1CFA		2CFA	
	Speedup	Taints	Speedup	Taints	Speedup	Taints
chess1	1.28	3	1.01	3	0.97	5
chess2	1.61	3	1.08	3	0.97	4
emoji-helper1	4.70	1	4.35	1	3.40	1
emoji-helper2	7.07	1	7.37	1	4.87	1
simple-translate	3.14	1	4.15	1	2.63	2
k-cup-deals	4.32	1	1.46	1	0.93	1
dateformat1	1.47	4	1.02	8	0.90	8
dateformat2	2.59	4	1.60	8	1.44	8
dateformat3	2.19	4	0.90	8	0.91	8
dateformat4	4.13	4	2.26	8	1.48	8
yallist1	1.18	7	1.03	60	1.07	60
yallist2	1.23	7	1.42	60	1.48	60
yallist3	9.36	7	14.41	60	13.16	60
balanced-match	0.88	20	0.97	420	1.02	420
url-join1	4.23	2	2.56	17	3.93	17
url-join2	4.27	2	2.60	17	3.99	17
Average:	3.35		3.01		2.70	

Table 4. Speedup results relative to a perfect incremental analysis, i.e., how close did we come to optimal speedup.

Benchmark	0CFA	1CFA	2CFA
chess1	0.20	0.31	0.34
chess2	0.17	0.19	0.27
emoji-helper1	0.59	0.53	0.60
emoji-helper2	0.77	0.78	0.86
simple-translate	0.99	0.98	1.00
k-cup-deals	0.37	0.47	0.83
dateformat1	0.34	0.26	0.30
dateformat2	0.54	0.41	0.46
dateformat3	0.54	0.31	0.48
dateformat4	1.00	0.82	0.76
yallist1	0.12	0.08	0.09
yallist2	0.15	0.11	0.14
yallist3	0.99	1.00	1.00
balanced-match	0.11	0.08	0.07
url-join1	0.96	0.90	1.00
url-join2	0.91	0.89	0.93
Average:	0.55	0.51	0.57

References

[1] balanced-match. https://github.com/juliangruber/balanced-match (2017)
[2] chess. https://bitbucket.org/rsb/chesscomnotifier/overview (2017)
[3] Chrome extension developers under a barrage of phishing attacks. https://tech.slashdot.org/story/17/08/11/221203/chrome-extension-developers-under-a-barrage-of-phishing-attacks (2017)
[4] dateformat. https://github.com/felixge/node-dateformat (2017)
[5] emoji-helper. https://github.com/johannhof/emoji-helper/blob/master/src/popup.js (2017)
[6] Google removes chrome extension used in banking fraud (2017), https://threatpost.com/google-removes-chrome-extension-used-in-banking-fraud/127469/
[7] k-cup-deals. https://addons.mozilla.org/en-US/firefox/addon/keurig-k-cup-deals/versions/ (2017)
[8] Malicious chrome extensions steal passwords & cpu power. https://duo.com/decipher/malicious-chrome-extensions-steal-passwords-and-cpu (2017)
[9] simple-translate. https://github.com/sienori/simple-translate/blob/master/simple-translate/background.js (2017)
[10] url-join. https://github.com/jfromaniello/url-join (2017)
[11] yallist. https://github.com/isaacs/yallist/blob/master/test/basic.js (2017)
[12] Arzt, S., Bodden, E.: Reviser: Efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In: ICSE 2014. pp. 288–298 (2014)
[13] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Transactions on software engineering **33**(9) (2007)
[14] Carroll, M.D., Ryder, B.G.: Incremental data flow analysis via dominator and attribute update. In: POPL '88. pp. 274–284. ACM (1988)
[15] Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In: International Conference on Computer Aided Verification. pp. 449–461. Springer (2005)
[16] Cooper, K.D., Kennedy, K.: Efficient computation of flow insensitive interprocedural summary information. In: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction. pp. 247–258 (1984)
[17] Cooper, K.D.: Interprocedural Data Flow Analysis in a Programming Environment. Ph.D. thesis, Rice University, Houston, TX, USA (1983), aAI8314924
[18] Ghodssi, V.: Incremental analysis of programs. Ph.D. thesis, University of Central Florida (1983)
[19] Hermenegildo, M., Puebla, G., Marriott, K., Stuckey, P.J.: Incremental analysis of constraint logic programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **22**(2), 187–223 (2000)
[20] Kapravelos, A., Shoshitaishvili, Y., Cova, M., Kruegel, C., Vigna, G.: Revolver: An automated approach to the detection of evasive web-based malware. In: USENIX Security Symposium (2013)
[21] Kashyap, V., Hardekopf, B.: Security signature inference for javascript-based browser addons. In: CGO. p. 219. ACM (2014)
[22] Krall, A., Berger, T.: Incremental flow analysis (1994)
[23] Kulkarni, S., Mangal, R., Zhang, X., Naik, M.: Accelerating program analyses by cross-program training. In: OOPSLA 2016. pp. 359–377. ACM (2016)
[24] Lee, H., Won, S., Jin, J., Cho, J., Ryu, S.: Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In: FOOL 2012. p. 96 (2012)
[25] Livshits, B., Guarnieri, S.: Gulfstream: Incremental static analysis for streaming javascript applications. Tech. rep., Microsoft Research (January 2010)
[26] Logozzo, F., Lahiri, S.K., Fähndrich, M., Blackshear, S.: Verification modulo versions: Towards usable verification. In: PLDI '14. pp. 294–304 (2014)
[27] Lu, Y., Shang, L., Xie, X., Xue, J.: An Incremental Points-to Analysis with CFL-Reachability, pp. 61–81. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
[28] Marlowe, T.J., Ryder, B.G.: An efficient hybrid algorithm for incremental data flow analysis. In: POPL '90. pp. 184–196 (1990)
[29] McPeak, S., Gros, C.H., Ramanathan, M.K.: Scalable and incremental software bug detection. In: FSE 2013. pp. 554–564 (2013)
[30] Mudduluru, R., Ramanathan, M.K.: Efficient Incremental Static Analysis Using Path Abstraction, pp. 125–139. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
[31] Nichols, L., Emre, M., Hardekopf, B.: Fixpoint reuse for incremental javascript analysis. Tech. Rep. 2019-02, University of California, Santa Barbara (March 2019), https://cs.ucsb.edu/research/tech-reports/2019-02
[32] Pollock, L.L., Soffa, M.L.: An incremental version of iterative data flow analysis. IEEE Transactions on Software Engineering **15**(12), 1537–1549 (1989)
[33] Rattan, D., Bhatia, R., Singh, M.: Software clone detection: A systematic review. Information and Software Technology **55**(7), 1165 – 1199 (2013)
[34] Souter, A.L., Pollock, L.L.: Incremental call graph reanalysis for object-oriented software maintenance. In: Proceedings IEEE International Conference on Software Maintenance. ICSM 2001. pp. 682–691 (2001)
[35] Szabó, T., Erdweg, S., Voelter, M.: Inca: A dsl for the definition of incremental program analyses. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 320–331. ACM (2016)
[36] Taly, A., Mitchell, J.C., Miller, M.S., Nagra, J., et al.: Automated analysis of security-critical javascript apis. In: 2011 IEEE Symposium on Security and Privacy. pp. 363–378. IEEE (2011)
[37] Tripp, O., Ferrara, P., Pistoia, M.: Hybrid security analysis of web javascript code via dynamic partial evaluation. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 49–59. ACM (2014)