Fixpoint Reuse for Incremental JavaScript Analysis (Extended Version)

Lawton Nichols, Mehmet Emre, and Ben Hardekopf

University of California, Santa Barbara, USA
{lawtonnichols,emre,benh}@cs.ucsb.edu

Abstract. When programs are frequently updated, the cost of statically analyzing those programs is multiplied by the number of program versions that must be analyzed. In cases such as analyzing JavaScript programs, the baseline analysis is costy, exacerbating this cost. One example of this is JavaScript-based browser addons which are continually updated and the addons must be repeatedly vetted for each update because of known instances of malicious code injection during updates.
Incremental analysis reduces this cumulative cost by using a previous version's solution to accelerate analysis of the current version. Modern incremental analyses are not applicable to dynamic programming languages because they assume an *a priori* control flow graph, which is not available. 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. A large part of our technique is a method for matching code between JavaScript program versions, a non-trivial problem for which existing techniques for non-JavaScript languages do not apply. 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.

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 [37,23,38].

However, a single-time static analysis is not sufficient when programs are continually updated with new versions. There are known instances where malicious code has been snuck into existing JavaScript programs during such updates [3].

To ensure safety and security, static analyses must be run on every version of a program, not just the first one. However, JavaScript is a highly dynamic and difficult language to analyze with precision, 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 update and 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 updates and 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 or (given the flow-graph model) trivially computable syntax mapping between the old and new program versions. Unfortunately, JavaScript programs do not have a simple flow-graph model, and in fact require extensive and expensive static analysis to compute precise control-flow and data-flow information. Thus, the existing works' assumptions do not hold and they 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 [13,22,34]; 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 $2\times$ of an *optimal incremental analysis* (which we define as an incremental analysis on a program version that is identical to the earlier version, 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. Previous analysis summaries 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.

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* [17,16,19]. 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 [33,14,30,21,29,35,12,25,28,36,31,15,32,24,20]. 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 simply computed from the respective flow-graphs. None of the precise incrementalization methods is immediately applicable to languages such as JavaScript.

"Incremental" Analysis of JavaScript

Livshits and Guarnieri [27] 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

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 in this section in order to convey the central concepts; Section 4 will go into more details within the context of our method's instantiation to a specific JavaScript analysis framework. We invite any reader without a background in program analysis to consult Section 6 for more information on abstract interpretation and taint analysis.

3.1 High-Level Summary

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, updated version of the program). We assume FP_{prior} is in the form of a map from program points to abstract states. The goal is to compute $FP_{\widehat{upd}}$, an over-approximation of FP_{upd} (the precise fixpoint analysis solution for P_{upd}).

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 ensuring that we visit every program point in P_{upd} at least once in order to guarantee a sound analysis.

Algorithm 1 shows a high-level view of the entire matching and reuse process. Section 4 describes exactly how we instantiate this generic algorithm in the case of JavaScript and the SAFE analysis framework.

Algorithm 1 Generic Fixpoint Reuse

1: **procedure** REUSE($P_{prior}, P_{upd}, FP_{prior}$)
2: **Input:** The two versions of the program, in some traversable form, and version P_{prior}'s fixpoint data structure
3: **Output:** $FP_{\widehat{upd}}$, a prepopulated fixpoint for version P_{upd}
4: Match program points, call sites, abstract addresses, and variable names of P_{prior} with those from P_{upd} using program similarity techniques
5: Populate $P_{prior} \rightarrow P_{upd}$
6: Populate $FP_{\widehat{upd}}$ using FP_{prior} and $P_{prior} \rightarrow P_{upd}$.
7: Analyze P_{upd} starting from $FP_{\widehat{upd}}$
8: **end procedure**

3.2 Example

To make this process more concrete, we provide a specific example based on taint-tracking program analysis. Consider the two program excerpts in Figure 1.

Version P_{prior} contains a function `f`, which is called with the argument `secret`, and we do not want the value of this secret variable to leak to the outside world. Assume that these snippets are part of a larger program.

```
function f(x) {
  for (...) {
    for (...) {
      // expensive computation
    }
  }

  return result;
}

var secret = 42;
var next = f(secret);
```

(a) Version P_{prior}

```
function f(x) {
  for (...) {
    for (...) {
      // same expensive computation
    }
  }
  output(x); // leak!
  return result;
}

var secret = 42;
var next = f(secret);
```

(b) Version P_{upd}

Fig. 1: Two programs, unalike in dignity

After running the analysis on version P_{prior}, we have a fixpoint solution FP_{prior} that maps every *(calling context, program point)* pair in P_{prior} to some abstract state. An abstract state will, for this analysis, map program variables and abstract heap locations to either *definitely tainted*, *definitely not tainted*, or *possibly tainted*. We can inspect the abstract state at the `output` statement to see if the value being output is tainted in any calling context. Suppose that the result is that there is no taint in this case.

Figure 1b shows an updated program version P_{upd} (which we can see by inspection does leak tainted data). Our incremental analysis will first use a program matching algorithm to find corresponding program points between the

two versions. In this case, it finds that the definition of `f` in P_{prior} matches the definition of `f` in P_{upd}, that a number of the basic blocks inside `f` match between P_{prior} and P_{upd}, and finally that the last two lines in each version also match.

The incremental analysis will then use this matching to transfer abstract states from FP_{prior} to $FP_{\widehat{upd}}$. As part of this matching and transfer, calling contexts and abstract heap locations from FP_{prior} will be renamed to the corresponding calling contexts and abstract heap locations appropriate to P_{upd}. The final analysis on P_{upd} will start from this seeded $FP_{\widehat{upd}}$ to compute the final fixpoint solution for P_{upd}. The analysis must be certain to visit every program point in P_{upd} at least once to guarantee a sound solution (e.g., by initializing the worklist with every program point rather than just the entry point). This requirement is due to the fact that otherwise the seeded $FP_{\widehat{upd}}$ may cause the analysis to prematurely converge at a pre-fixpoint solution.

The incremental analysis correctly concludes that there is a leak in the updated program. With reuse, the taint analysis on version P_{upd} could reuse the analysis results for a vast majority of the program and thus converge much faster than a from-scratch analysis. While this example is trivial, we have achieved good results and significant speedups using this method on real-world JavaScript code that runs in browsers and/or servers.

3.3 On Program Matching

Computing the map $P_{prior} \rightarrow P_{upd}$ is an important part of the process that can have extreme effects on the efficacy of the incremental analysis. 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 in this case, but it can simply compute the information in the same way as a from-scratch analysis.

The second case, however, is by far the worst case and demonstrates 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 polluting precision. Thus, it is far better to fail to match a program point than it is to incorrectly match a program point. This means that 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*.

4 Fixpoint Reuse for SAFE

Our prototype implementation is built on top of SAFE version 2.0. The SAFE JavaScript analysis framework [26] 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.).

In order to reuse analysis results, we must therefore create a correspondence mapping between programs at the level of SAFE's IR. In this section we describe its constituent pieces, along with the different matching methods we implemented and evaluated against.

4.1 Functions, Blocks, and Instructions, Oh My!

SAFE programs are divided into three main categories: functions, blocks, and instructions. We explain them via an example. Consider the JavaScript program in Figure 2a, made up of two functions and a free-standing statement that calls one of those functions. This program is translated into SAFE's intermediate program as Figure 2b.

At both the textual- and the data structure-level, the program is represented as a hierarchy of three entities: functions, blocks, and instructions. Any statements that occur outside of a function are gathered together in the `top-level` function, which serves as the entry point to the translated IR version of the program and its subsequent analysis. For example, there are eight separate blocks in the `top-level` function—Entry[-1], Block[0], Call[1], AfterCall[2], AfterCatch[3], Block[4], Exit[-2], and ExitExc[-3]. There are nine instructions inside Block[0], which prepares for the call to `isEven`, and there is just one instruction in the Call[1] block, which performs the actual call to the function.

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 (in the IR, anything that begins with a "#") 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 it is important that we match with high confidence if we want any hope of making our analysis reuse method efficient and accurate.

```
function isEven(x) {
  if (x == 0) {
    return true;
  } else {
    return isOdd(x-1);
  }
}

function isOdd(x) {
  if (x == 0) {
    return false;
  } else {
    return isEven(x-1);
  }
}

var b = isEven(42);
```

(a) A JavaScript program

```
function[0] top-level {
    Entry[-1] -> [0]

    Block[0] -> [1], ExitExc
  [0] isEven := function (1) @ #4, #5
  [1] isOdd := function (2) @ #9, #10
  [2] noop(StartOfFile)
  [3] <>obj<>15 := @ToObject(isEven) @ #11
  [4] <>temp<>16 := 42
  [5] <>arguments<>17 := allocArg(1) @ #12
  [6] <>arguments<>17["0"] := <>temp<>16
  [7] <>fun<>18 := @GetBase(isEven)
  [8] <>this<> := enterCode(<>fun<>18)

    Call[1] -> ExitExc
  [0] call(<>obj<>15, <>this<>, <>arguments<>17)

    AfterCall[2] -> [4]

    AfterCatch[3] -> ExitExc

    Block[4] -> Exit, ExitExc
  [0] b := <>Global<>ignore1
  [1] noop(EndOfFile)

    Exit[-2]

    ExitExc[-3]

}

function[1] isEven {
  ...
}

function[2] isOdd {
  ...
}
```

Key
- Function ID
- Block ID
- Abstract address

(b) The same program converted to SAFE IR

Fig. 2: SAFE's intermediate representation for JavaScript programs

4.2 Function Matching

Our function matching algorithm is based on an edit-distance calculation, as shown in Algorithm 2. The algorithm is parameterized by a function Criteria that determines 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 as shown in Table 1 in order to evaluate which combination of distance criteria worked best. Given the distances, the algorithm matches those functions with the best distance score that is under our empirically-calculated threshold.

Algorithm 2 works under the assumption that functions between program versions may be nested differently but generally still appear in the same order. We took inspiration from Revolver, a work which found success using a longest common subsequence algorithm to find similar pieces of malware among JavaScript programs [22]. Longest common subsequence is a specific instantiation of the more general problem of edit distance, and so we chose to design our matching algorithms around edit distance calculations—it plays a part in our block and instruction matching methods as well. Matches can be extracted from the algorithm's resulting table.

Algorithm 2 Edit Distance Function Matching

procedure Edit-Distance-Match-Functions($\overrightarrow{funcs_{curr}}, \overrightarrow{funcs_{upd}}$, Criteria)
Input: Functions from version P_{prior}, functions from version P_{upd}, and a function Criteria that scores functions based on similarity
Output: A list of pairs of functions that have been deemed similar
  matchingFunctions $\leftarrow \emptyset$
  $M \leftarrow$ Parameterized-Edit-Distance($\overrightarrow{funcs_{curr}}, \overrightarrow{funcs_{upd}}$, Criteria)
  Inspect the score matrix M, and populate *matchingFunctions* with the function pairs that were successfully matched
  return *matchingFunctions*
end procedure

Function Similarity Scoring Criteria. Table 1 shows the different kinds of function criteria that we evaluate against. These are different combinations of differences based on function position and based on instructions contained within the functions. We chose the two axes of position-based and instruction-based matching after manually inspecting the version differences among our benchmarks: quite often we discovered that the functions appeared in the same order, and that the instructions for the most part were identical. We also noticed that there were cases of the same function body appearing in multiple places, so that led us to combine the two areas in different ways to determine which combination was best. The criteria are in the form of distance functions, so a higher number indicates a larger difference. We combine features using the geomean.

Table 1: Different function matching criteria

Name	Criteria
Position-only	Distance between function IDs, distance between function line numbers
Instruction-only	Difference between number of instructions, (Size of the larger multiset of identifiers that occur in each function) – (Number of common identifiers occurring in both the functions)
Combined	Combination of Position-only and Instruction-only
Staged	Instruction-only, with ties broken by Position-only

4.3 Block Matching

Our chosen block matching algorithm also uses an edit distance calculation. Edit distance-based block matching is similar to Algorithm 2, and the blocks of two functions are matched using edit distance in the order in which they appear in the SAFE IR. This choice is based on the assumption that that changes to functions will not drastically affect the analysis results, and so matching as many blocks as we can will be helpful when we transfer old analysis computations. The best scoring blocks that match under a threshold are returned.

Block Similarity Scoring Criteria. The block criteria we chose is a combination of the following:

- Block type (Normal, Call, etc.),
- Instruction count difference, and
- (Size of the larger multiset of identifiers that occur in each block) - (Number of common identifiers occurring in both the blocks).

We save the corresponding blocks, because Call blocks are used in abstract state calling contexts.

4.4 Instruction Matching

For each matched pair of blocks, we once again perform a 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 the generated numerical suffixes). We save the corresponding allocation sites and variables that appear as the left-hand side of assignment instructions, as they will need to be remapped between abstract heaps.

4.5 Fixpoint Reuse

After a successful program matching effort we must use this information to remap results from FP_{prior} into the new $FP_{\widehat{upd}}$ before the incremental analysis

begins its calculation. We keep this section high-level; the details are tedious but straightforward.

Figure 3 shows a high-level version of the data structures we work with. SAFE contains a map that keeps track of the saved abstract state for every ControlPoint, which is a (Block, Context) pair. Contexts keep track of which functions were called immediately preceding the current one, and can contain zero or more call sites, which are themselves Blocks (specifically, the Call blocks where the function call to the current function or one of its predecessors took place). For example, if we start out from the top level of our program, and call a function `foo`, the context will change from $\emptyset$ to the list [`foo`], assuming our context sensitivity is greater than zero. So, we must remap each block using our saved correspondence mapping generated during the matching phase; it is for this purpose that we keep track of corresponding blocks.

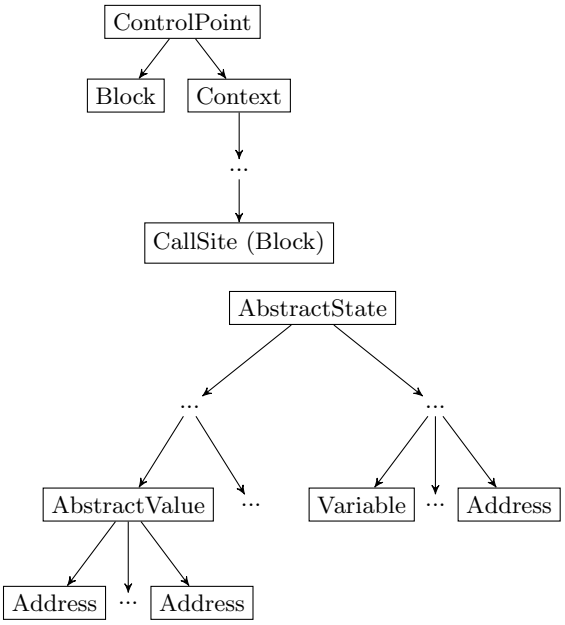


Fig. 3: Reuse occurs inside of the ControlPoint and AbstractState data structures. This figure shows the general layout of these structures, and the leaves are things that we must map over between versions. For example, the abstract Addresses will often change with program versions.

For the abstract states, we must traverse and find any addresses and local variable names that we know how to map over. Local variables are held in a specific part of the abstract state, while abstract addresses are spread everywhere (though mainly exist in the heap and abstract JavaScript objects). It is for this purpose that we keep track of corresponding instructions.

5 Evaluation

In this section we evaluate the efficacy of fixpoint reuse in terms of performance and precision. Because we're 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 2. These benchmarks were chosen from a set of similar programs because SAFE can completely model their code and analyze them using a reasonable amount of resources (we are limited in our benchmark selection by SAFE's capabilities). Following previous work on analyzing browser add-ons [23], 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.

Table 2: 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

The actual analysis that we perform on these benchmarks is a taint analysis implemented using the SAFE JavaScript analysis infrastructure, suitably mod-

ified 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.

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. These results are in Section 5.3, and we study the sizes of the reused fixpoints in Section 5.4.

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 shown in Table 3.

5.2 Controlled Distance Study

Table 4 contains information on our handmade benchmarks: they are versions of the v8 Navier-Stokes (Table 4a) and the Richards (Table 4b) benchmarks with statements deleted. 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

[1] `https://anonfile.com/6ek9W1gab0/fixpoint-reuse-artifact_zip`

Table 3: Best possible speedups. Exact times can be calculated using Table 6.

Benchmark	0CFA (×)	1CFA (×)	2CFA (×)
chess1	6.30	3.23	2.84
chess2	9.59	5.66	3.61
emoji-helper1	7.98	8.17	5.64
emoji-helper2	9.15	9.47	5.67
simple-translate	3.17	4.24	2.60
k-cup-deals	11.58	3.12	1.12
dateformat1	4.30	3.93	3.00
dateformat2	4.84	3.92	3.12
dateformat3	4.05	2.95	1.92
dateformat4	4.04	2.77	1.96
yallist1	9.85	13.05	11.93
yallist2	8.37	13.44	10.84
yallist3	9.44	13.36	11.55
balanced-match	7.92	12.58	14.46
url-join1	4.41	2.85	3.93
url-join2	4.68	2.93	4.30
Average	6.85	6.58	5.53

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 4 shows the results of our handmade benchmarks. 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.

Figure 4a shows the results for the Richards benchmarks. This 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. Figure 4b shows the results for the Navier-Stokes benchmarks. This benchmark consists of a small number of large functions. The original benchmark's fixpoint took 7,060 iterations to converge, there were 562 unique program points visited, and there were 26 loops. For both benchmarks, the chosen taints were calculated precisely for all version pairs.

Both sets of benchmarks tend to degrade in performance as the difference between version pairs increases, but the Richards benchmark appears to be more amenable to reuse and therefore has more to lose as the distance increases. Given

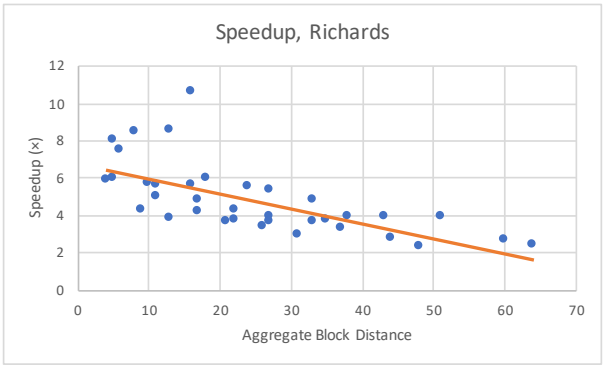
Table 4: Handmade Benchmarks

(a) Navier-Stokes (v8 lines of code: 398)

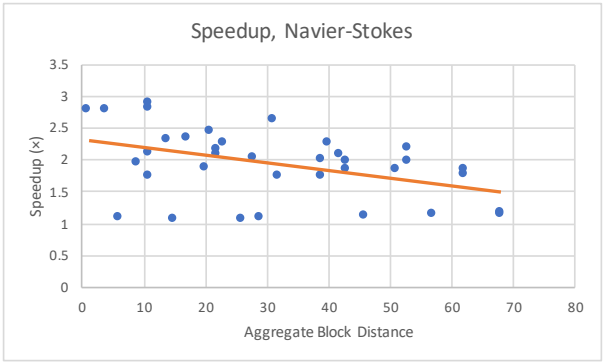
Versions (A–B)	Diff	Distance
v0–v8	35+/4–	68
v1–v8	32+/4–	62
v2–v8	27+/3–	53
v3–v8	20+/3–	43
v4–v8	17+/2–	39
v5–v8	11+/2–	22
v6–v8	8+/2–	11
v7–v8	2+/0–	1

(b) 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



(a) Richards



(b) Navier-Stokes

Fig. 4: Handmade Results

the statistics in the previous paragraph, compared with the Navier-Stokes benchmark set, the Richards benchmark set has more iterations to save by reusing information, fewer program points to visit at least once, and significantly fewer loops through which any updated information must be propagated. This experiment helps to give insight into which kinds of programs see better fixpoint reuse performance, while also highlighting that some amount of improvement is usually possible as long as the changes are not too drastic. Even in the face of unrecoverable program differences (see the line of dots hovering above the $1\times$ speedup in Figure 4b), for these benchmarks our method does not do worse than a from-scratch analysis.

Unmatched instructions Figure 5 provides insight into the abilities of our matching algorithm by showing the total number of instructions that could not be matched across all the different versions of the Richards handmade benchmarks. The Navier-Stokes results are similar.

As the textual difference between programs increases, it becomes more difficult to match functions and blocks—this difficulty culminates in the algorithm's inability to match individual instructions inside of blocks. The larger the gap between handmade benchmark versions, the more changes exist in the code, and the number of unmatched instructions reflects this. Matching is not exact, and so there is one outlier in the v0–v2 version pair, but the results show that our matching method keeps unmatched instructions to a minimum and performs well in the vast majority of cases.

For the instructions that could be matched, there is still a possibility that they were matched incorrectly—discovering errors in instruction matching would require manual effort, but we believe the lack of extra taints demonstrates that our edit distance-based matching is precise.

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. Table 5 shows the times, and they are all quite small. For the longer-running analyses this number is completely negligible.

Baseline results Table 6 shows the results for running the static analysis on the updated version of each benchmark from scratch (i.e., with fixpoint reuse turned off). The number of taints output is the sum of all tainted sources for a given tainted sink state—note that states can become duplicated when context sensitivity increases, and that is why the number increases.

Incremental Results Table 7 shows the results of reuse for each different context sensitivity level. We find that all taints are carried over with very little

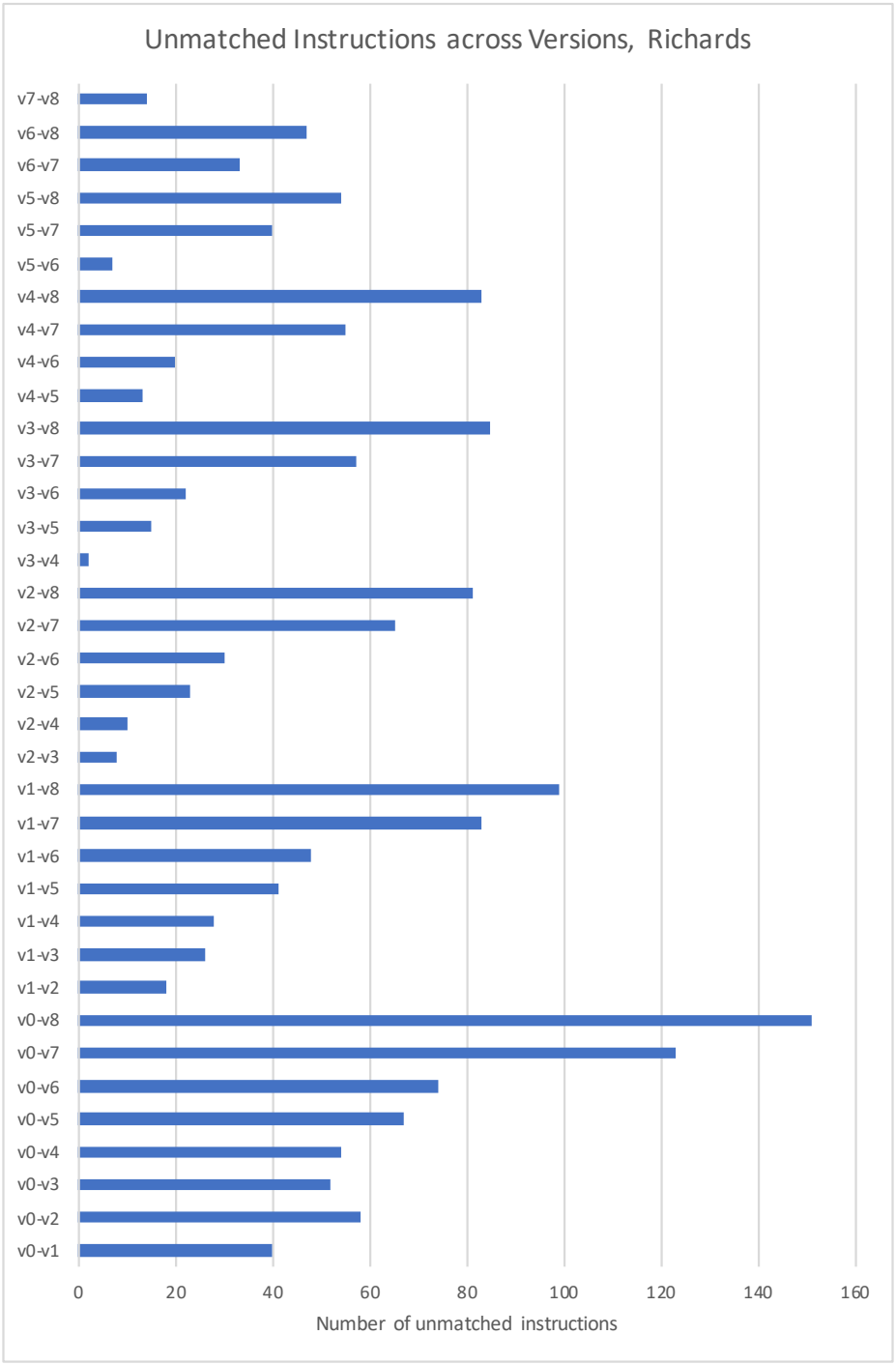


Fig. 5: Number of unmatched instructions across the Richards handmade benchmarks.

Table 5: Correspondence map creation times.

Benchmark	Time (s)
chess1	1.03
chess2	1.07
emoji-helper1	4.70
emoji-helper2	4.82
simple-translate	1.18
k-cup-deals	2.24
dateformat1	0.82
dateformat2	0.75
dateformat3	0.77
dateformat4	0.84
yallist1	3.92
yallist2	4.27
yallist3	4.33
balanced-match	0.42
url-join1	0.55
url-join2	0.55

Table 6: Baseline results.

Benchmark	0CFA		1CFA		2CFA	
	Time (s)	Taints	Time (s)	Taints	Time (s)	Taints
chess1	39.46	3	22.20	3	24.28	5
chess2	71.77	3	39.05	3	39.01	4
emoji-helper1	213.90	1	135.78	1	84.55	1
emoji-helper2	251.39	1	150.80	1	95.04	1
simple-translate	13.60	1	17.34	1	14.29	2
k-cup-deals	86.89	1	30.58	1	20.45	1
dateformat1	52.28	4	56.71	8	50.92	8
dateformat2	67.06	4	60.39	8	55.52	8
dateformat3	66.60	4	42.74	7	34.34	7
dateformat4	68.92	4	44.25	7	35.76	7
yallist1	843.39	7	1443.23	60	2084.17	60
yallist2	857.51	7	1432.67	60	2033.51	60
yallist3	823.98	7	1429.20	60	2107.08	60
balanced-match	249.31	20	523.17	420	2397.87	420
url-join1	41.11	2	29.83	17	112.95	17
url-join2	41.60	2	30.42	17	114.35	17

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.

Table 7: Results, relative to from-scratch analysis. Exact times can be calculated using Table 6.

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	

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 8 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. Due to natural variation in analysis times, some results were slightly above 1.0—we capped those results at 1.0 to paint a more accurate picture. 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.

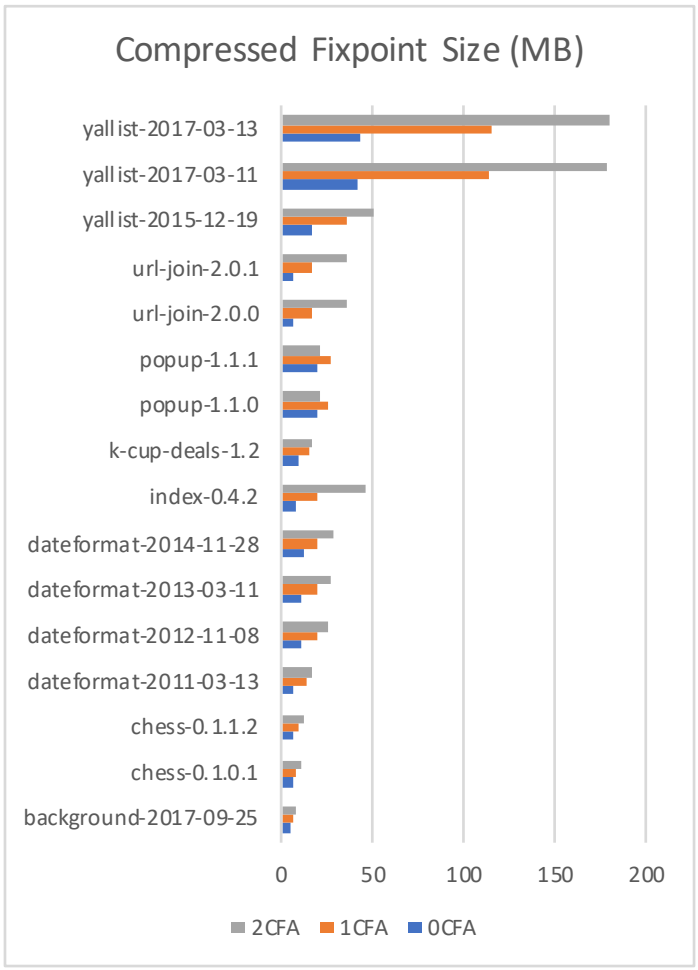


Fig. 6: Size of gzip-compressed fixpoints in megabytes.

Table 8: Speedup results, relative to a perfect incremental analysis. Exact times can be calculated using Table 6.

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

5.4 Size of Saved Fixpoints

Figure 6 shows the sizes of each fixpoint after compressing with the `gzip` compression utility. The longer-running benchmarks compute more information during the analysis, and are therefore larger. An increase in context sensitivity also causes more information to be saved, and this increases the size of each fixpoint as well. As these files can get large, an interesting area of future investigation could be investigating how to take advantage of the semantic struture of fixpoints to aid further compression.

6 Background

In this section we provide some background on abstract interpretation and taint analysis.

6.1 Abstract Interpretation

Abstract interpretation [18] discovers invariants about a program by running it in an abstract way. For example, assume that the computation of $2 + 3$ was too difficult to compute—one way of making this easier computationally is to get rid of the numbers, and instead calculate $\mathit{positive} + \mathit{positive}$—a table lookup would provide the final answer of *positive*. We can capture this process with

a function $\alpha : \mathbb{Z} \to \mathit{PosNegZero}$, where *PosNegZero* is the powerset of the set $\{\mathit{positive}, \mathit{negative}, 0\}$; it is an example of an *abstract domain*.

positive, however, is not the same as 5—it is an *abstraction*. Information is clearly yet deliberately lost when we move to an abstraction, and there is much work on the fine line between giving up too much information and just enough to solve the original problem. Information is ordered in the logical way: $\{\mathit{positive}\}$ is clearly more precise than $\{\mathit{positive}, 0\}$, so we write this fact as $\{\mathit{positive}\} \sqsubseteq \{\mathit{positive}, 0\}$ in this new abstract sense, so less precise things are abstractly *bigger* than more precise things—intuitively, such imprecise values "believe" that they are more things.

Programming languages are more complicated than simple arithmetic expressions: we must figure out how to "abstractly" run the computations involved during function calls and complicated loops and numerical operations. And, we must do all this while preserving the key property that all program analyses must have: the analysis must halt, even if the program does not. We want some information about a program, and we accept that we must settle for some information loss, but we don't want to wait a lifetime to get the final answer. A more complicated analysis that is essential for JavaScript analysis is called control-flow analysis: instead of generating values of variables, it creates a control flow graph for the analyzed program.

Fortunately, we are not forced to reinvent the wheel to implement a program analysis. One piece of machinery that gets us most of the way there is an interpreter for a programming language—it has most of the features that we desire, but it is too precise. We can implement an abstract interpreter on top of a regular (the technical term is *concrete*) interpreter fairly easily (see, e.g., [39]) after one has chosen the abstract domains to use.

Interpreters (and therefore abstract interpreters) often have a lot of moving parts—especially for a language like JavaScript, where values can be of many different types all at once, functions are first-class objects, inheritance is modeled with prototypes, etc. Abstract heaps are often used to keep track of data that has been allocated, and there is also a stack of functions that have been called and pointers to where they should return to. None of this information goes away during the transition to an abstract interpreter.

Nondeterminism often plays a large role in abstract interpretation, due to lost precision—this can cause loops to arise where they would not appear otherwise. As an example of lost precision and nondeterminism, consider the following program statement:

```
if (1 <= 2)
  X();
else
  Y();
```

A concrete interpreter would immediately conclude that only the true branch will ever be executed, but consider the abstract interpreter with the *PosNegZero* numerical abstract domain from above: it would instead ask itself about the truth or falsity of the expression $\mathit{positive} \leq \mathit{positive}$, and the correct answer is "I don't know", since `4 <= 3` looks exactly the same as `1 <= 2` from its viewpoint. Thus

the abstract interpreter returns `true` and `false` for this expression, runs *both branches*, and must find a way to *combine* the information that it calculates for each branch to use as the result of the entire `if/else` expression.

This combination of imprecise values is key to our analysis reuse: maybe the program changes, and `X()` does something completely different while `Y()` remains the same—could we use our old information about the abstract result of `Y()` again? Our method revolves around populating abstract states with previously-computed values.

A term that appears often in this paper is *context sensitivity*, which describes how an analysis keeps track of where functions that are called should return. It is infeasible (and in programs with call stacks of indeterminate depth, impossible) to keep track of the entire call stack in an abstract interpreter, and so many analyses settle for saving only the very top of the stack. Saving one function on the top of the call stack is abbreviated as 1CFA, two is 2CFA, and saving nothing is called either 0CFA or *context insensitive*. These different context sensitivities make a difference in the precision and performance of an analysis, and this difference translates into the realm of reuse that we explore.

Abstract interpreters often run what is called a worklist algorithm, which keeps track of which locations in the program to visit and calculate abstract results for—initially, the first line of the program is inserted as the initial state, because the analysis has not yet saved any information about its execution. The analysis halts when the information it knows about each state does not change. When the analysis stops and the information converges, we say that it has reached a *fixpoint*. We refer to this conglomerate of saved information about each abstract state as the "fixpoint" of the entire analysis.

If we give an analysis some starting information from a previous program version, it has the potential to terminate more quickly, though there is the risk of having too much imprecision. That is the challenge we overcome in this paper.

One important property that a analysis should have is soundness: a sound analysis overapproximates the results of any concrete run of a program and does not leave out a possible behavior—we ensure that our analysis reuse method retains the soundness of the underlying analysis.

6.2 Taint Analysis

As the name indicates, a program taint is something that is undesirable. Taints have to do with data dependence, i.e., which values get where. For example, consider the following code:

```
var x = 5;
var y = x + 7;
var z = y * 8 + 42;
```

We say that `x` *flows to* `z` (and also `y`). If the number `5` was a confidential value that we did not wish to pass around to too many places, we would say that it is a tainted *source* and try to track where this information *flows*. Instead

of propagating actual numbers or positivity information, this analysis tracks information about the taintedness of a given expression.

Taint tracking is a common program analysis, and the goal is to find whether any tainted sources reach any undesirable program locations, or *sinks*. A more high-level example of a source and a sink is that of a system password and a network request.

To evaluate our analysis reuse method, we implement taint analysis on top of an existing JavaScript abstract interpreter, and we use this analysis to evaluate the precision of our worklist reuse algorithm. We want the reused analysis to not return too many extra sinks that an analysis that started from scratch would not return, for some definition of "too many".

This may sound too good to be true, so we briefly provide some insight on why this goal of precise reuse without any pruning is attainable. Consider once more the example of the imprecise `if` statement from the previous subsection, and imagine again that `X()` is changed drastically while `Y()` remains the same. We will carry over the taints from `Y()` in the reused analysis, since it has an exact correspondence in the new program, but we will also remember any taints that occurred inside `X()`, and at first glance this may be interpreted as a bad thing. But, the key point is that we assume that the old program was "accepted" or already considered safe, so any "incorrect" taints will in fact not pollute the output. Our evaluation confirms this.

7 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. A major aspect of this technique is how to match program points between two program versions, for which we have leveraged techniques from clone detection. 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.

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 (8 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. https://threatpost.com/google-removes-chrome-extension-used-in-banking-fraud/127469/ (8 2017)
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 (11 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. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL '77. pp. 238–252 (1977)
19. Ghodssi, V.: Incremental analysis of programs. Ph.D. thesis, University of Central Florida (1983)
20. 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)
21. Kalman, D., Pistoia, M., Podjarny, G., Tripp, O., Weisman, O.: Incremental static analysis (Mar 2012), https://www.google.com/patents/US20120054724, US Patent App. 12/873,219
22. 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)
23. Kashyap, V., Hardekopf, B.: Security signature inference for javascript-based browser addons. In: CGO. p. 219. ACM (2014)

24. Krall, A., Berger, T.: Incremental flow analysis (1994)
25. Kulkarni, S., Mangal, R., Zhang, X., Naik, M.: Accelerating program analyses by cross-program training. In: OOPSLA 2016. pp. 359–377. ACM (2016)
26. 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)
27. Livshits, B., Guarnieri, S.: Gulfstream: Incremental static analysis for streaming javascript applications. Tech. rep., Microsoft Research (January 2010)
28. Logozzo, F., Lahiri, S.K., Fähndrich, M., Blackshear, S.: Verification modulo versions: Towards usable verification. In: PLDI '14. pp. 294–304 (2014)
29. 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)
30. Marlowe, T.J., Ryder, B.G.: An efficient hybrid algorithm for incremental data flow analysis. In: POPL '90. pp. 184–196 (1990)
31. McPeak, S., Gros, C.H., Ramanathan, M.K.: Scalable and incremental software bug detection. In: FSE 2013. pp. 554–564 (2013)
32. Mudduluru, R., Ramanathan, M.K.: Efficient Incremental Static Analysis Using Path Abstraction, pp. 125–139. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
33. Pollock, L.L., Soffa, M.L.: An incremental version of iterative data flow analysis. IEEE Transactions on Software Engineering **15**(12), 1537–1549 (1989)
34. Rattan, D., Bhatia, R., Singh, M.: Software clone detection: A systematic review. Information and Software Technology **55**(7), 1165 – 1199 (2013)
35. 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)
36. 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)
37. 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)
38. 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)
39. Van Horn, D., Might, M.: Abstracting abstract machines. In: ICFP. pp. 51–62. ICFP '10, ACM, New York, NY, USA (2010)