Mimic: Computing Models for Opaque Code

Stefan Heule
Stanford University
Stanford, CA, USA
sheule@cs.stanford.edu

Manu Sridharan
Samsung Research America
Mountain View, CA, USA
m.sridharan@samsung.com

Satish Chandra
Samsung Research America
Mountain View, CA, USA
schandra@acm.org

ABSTRACT

Opaque code, which is executable but whose source is unavailable or hard to process, can be problematic in a number of scenarios, such as program analysis. Manual construction of models is often used to handle opaque code, but this process is tedious and error-prone. (In this paper, we use model to mean a representation of a piece of code suitable for program analysis.) We present a novel technique for automatic generation of models for opaque code, based on program synthesis. The technique intercepts memory accesses from the opaque code to client objects, and uses this information to construct partial execution traces. Then, it performs a heuristic search inspired by Markov Chain Monte Carlo techniques to discover an executable code model whose behavior matches the opaque code. Native execution, parallelization, and a carefully-designed fitness function are leveraged to increase the effectiveness of the search. We have implemented our technique in a tool MIMIC for discovering models of opaque JavaScript functions, and used MIMIC to synthesize correct models for a variety of array-manipulating routines.

Categories and Subject Descriptors

I.2.2 [**Automatic Programming**]: Program synthesis

Keywords

Opaque code, program synthesis, MCMC, model generation, JavaScript

1. INTRODUCTION

We consider the problem of computing *models* from *opaque* code. By *opaque*, we mean code that is executable but whose source is unavailable or otherwise difficult to process. Consider the case of a JavaScript runtime, which provides "native" implementation of a large number of utility functions, such as array functions. Natively-implemented methods are opaque to an analysis tool built to analyze JavaScript sources, and are a hindrance to effective analysis because the tool must make either unsound or overly conservative assumptions on what those native methods might do. This same situation arises for many languages and runtimes.

Opacity also is a concern when a third-party library is distributed in deliberately obfuscated form that hinders static analysis. In JavaScript, for example, an obfuscator might replace access to fields by computed names, e.g., transforming `y = x.foo` to `p = ["o","fo"]; y = x[p[1]+p[0]]`. Such changes can foil a static analyzer's ability to reason precise about data flow through the heap.

Models provide an alternate, easy to analyze representation of the opaque code. Sometimes, models can be written by hand; however, this is tedious and error prone, and requires understanding of how the opaque code works. A need for better, automated ways of creating models has been argued in the literature [10, 19, 7, 11].

In this paper, we show a new, automatic way of creating models for opaque functions, based on a novel search-based program synthesis strategy. We observe that the behavior of an opaque function can often be indirectly observed by intercepting accesses to memory shared between the client and the opaque code. (Here, "shared" is in the sense of an object that is passed as a parameter to a called function and has nothing to do with concurrency.) In most common dynamic languages (e.g., JavaScript, Lua, Python, Ruby), interception of accesses to shared objects can be achieved using indirection via proxy objects (discussed in more detail in Section 4). In this paper, we show that, surprisingly, it is in fact possible to generate useful models of opaque functions *based solely on observation of these shared memory accesses*.

Given a set of inputs for an opaque function, our technique collects traces of (shared) memory accesses to those inputs by running the function against those inputs and recording the intercepted accesses. It then carries out a random search, inspired by Markov Chain Monte Carlo (MCMC) sampling, to synthesize from scratch a function whose execution results in the same sequence of reads and writes. Our strategy is a "generate-and-test" strategy, leveraging efficient native execution—simply running it—to test the quality of candidate models. Comparison of quality of models is done using a carefully designed fitness function that takes into account the degree to which a model matches the available traces. Native execution lets the technique run tens of thousands of trials per minute, and it yields models that are in fact concrete code. Thus, our models are agnostic to whatever abstraction a program analysis wishes to employ. Figure 2 shows the model that our approach recovers for the JavaScript `Array.prototype.shift` method. Notice that the model

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FSE '15, Bergamo, Italy
Copyright 2015 ACM X-XXXXX-XX-X/XX/XX ...$15.00.

includes complex control-flow constructs in addition to the elementary statements.

Related approaches The problem of generating an implementation from a trace of actions is not new, but to the best of our knowledge, our technique is new. Closely related work [10, 6] in this area has used the concept of a *version-space algebra* to search through a space of candidate programs. In version-space algebra, a domain-specific data structure encodes all programs that are consistent with the traces seen so far. When a new trace is presented, the data structure is adjusted to eliminate those programs that would be inconsistent with this new trace. The final space can be ranked using domain-specific insights. The success of the version-space technique depends on careful design both of the space of domain-specific programs and of the data structure that represents them, in such a way that the knowledge embedded in a new trace can be factored in efficiently [6].

While the version-space algebra approach has been very successful in specific domains such as spreadsheet manipulation, its success has not been shown on general-purpose programs. The work by Lau et al. [10] handles only the situation in which the complete trace with the state of *all* variables is given at each step, along with the knowledge of the program counter; this simplifies the search space considerably. Given our execution traces, which contain only shared memory accesses and no program counter information, there is not clear way to represent all consistent programs in a compact data structure that can be efficiently updated for new traces.

Other recent work uses an "oracle-guided" approach to model generation, based on constraint generation and SMT solvers [15, 8]. The generated constraints allow for a model to be constructed from pre-defined "building blocks" or "components," and they ensure the model is consistent with input-output examples obtained from running the opaque code. Here, the limitation is that the set of building blocks must be chosen very carefully to make the constraint problem tractable [15]—it is unclear if such an approach could scale to generate models as complex as that of Figure 2 without excessive hand-tuning.

The alternative, then, is to adopt a generate-and-test strategy, and the important question is how to organize the generation of candidates so as to not have to evaluate enormous spaces (even if bounded) of programs exhaustively. In this regard, recent work [4] has shown the promise of genetic algorithms to synthesize patches to fix buggy programs. In that work, the operations of mutation and crossover are applied to an existing defective program that has both passing and failing test inputs. Mutation is applied by grafting existing code exercised in failing inputs, and this contains the search space to manageable size. Since our problem is to synthesize code from just the execution traces, we cannot readily adopt this method: there is no place to graft from. Nevertheless, this work has strongly inspired our own work in its use of statistical techniques to explore an enormous search space of code fragments.

For our setting, an approach inspired by MCMC sampling seems to be an effective search strategy, and this is borne out by our results.

Overview of Results We have implemented our approach in a tool called Mimic for JavaScript. It collects traces by wrapping function parameters with JavaScript proxy objects (see Section 4). We have found Mimic to be surprisingly effective in computing models of several of the array-manipulating routines in JavaScript runtime. In fact, the models generated by Mimic have occasionally been of higher quality than the hand-written models available with WALA, a production-quality program analysis system [20]. When it succeeded, Mimic required, on average, roughly 60 seconds per method, evaluating hundreds of thousands of candidates in the process. The technique is easily parallelized, and in our evaluation we leveraged a 28-core machine for improved performance. Mimic was just as capable in extracting models from obfuscated versions of the same library methods written in JavaScript. Our implementation is available at https://github.com/Samsung/mimic.

```
function shift(arg0) {
  var n0 = arg0.length
  if (n0) {
    var n1 = arg0[0]
    for (var i = 0; i < (n0-1); i += 1) {
      var n2 = (i+1) in arg0
      if (n2) {
        var n3 = arg0[i+1]
        arg0[i] = n3
      } else {
        delete arg0[i]
      }
    }
    delete arg0[i]
    arg0.length = i
    return n1
  } else {
    arg0.length = 0
  }
}
```

Figure 2: Model of `Array.prototype.shift` generated by our tool.

As with any search-based technique, there are limitations (see Section 5.3 for a detailed discussion). Our approach is currently limited to discovering models of functions that primarily perform data flow, with relatively simple logical computation. We cannot generate a model for a complex mathematical function (e.g., sine), since our trace of shared memory accesses does not expose the complex calculations within the function. Nevertheless, we believe our approach can still be usefully applied in a variety of scenarios.

Contributions This paper makes the following contributions:

- We introduce the problem of computing models for opaque code, and present a novel search-based synthesis algorithm to address this problem. To our knowledge, no other technique can find models for opaque code with the minimal information that our algorithm requires.
- We describe a prototype synthesizer Mimic for JavaScript functions based on use of proxy objects, and show that it can successfully synthesize accurate models for a variety of array-manipulating routines.

2. OVERVIEW

In this section, we give an overview of our synthesis technique Mimic, using an example function from the JavaScript standard library.

Consider JavaScript's built-in `Array.prototype.shift` function. For a non-empty array, `shift` removes its first element e

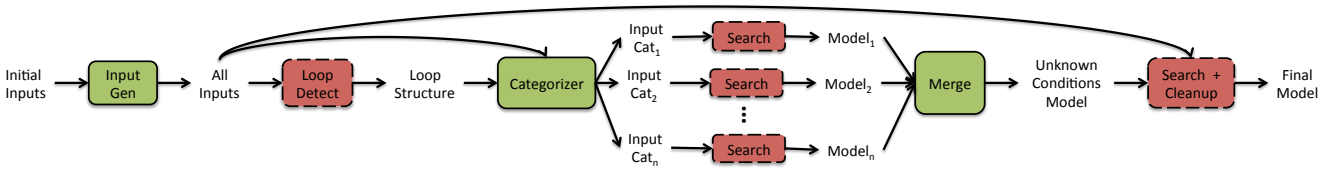

Figure 1: Overview of Mimic phases. The green phases (with solid outlines) are deterministic, while the red phases (with dashed outlines) incorporate randomness. We call the procedure corresponding to the whole figure MimicCore.

(at index 0), shifts the remaining values down by one index (so the value at index 1 is shifted to 0, 2 to 1, etc.), and finally returns e, e.g.:

```
var arr = ['a','b','c','d'];
var x = arr.shift();
// x is 'a', and arr is now ['b','c','d']
```

For an empty array, `shift` returns the value `undefined`. Mimic is able to generate a model for `shift`, shown in Figure 2, that captures the above behaviors based solely on collected execution traces, with no access to a source representation. Note that the method also works for so-called sparse arrays, which may have certain elements missing.

In the remainder of this section, we outline the steps taken by Mimic to generate this model. These steps are illustrated in Figure 1.

Generating inputs and traces. Our approach begins by generating a set of execution traces for the function in question, based on some initial inputs provided by the user. An execution trace includes information about the memory operations performed by the function on any objects reachable from its parameters, and also what value was returned. Mimic uses proxy objects to collect such traces for JavaScript functions (details in Section 4). For the `shift` function, given the input `['a','b','c','d']`, we obtain the following trace:

```
read field 'length' of arg0 //  4
read field 0 of arg0 // 'a'
read field 1 of arg0 // 'b'
write 'b' to field 0 of arg0
read field 2 of arg0 // 'c'
write 'c' to field 1 of arg0
read field 3 of arg0 // 'd'
write 'd' to field 2 of arg0
delete field 3 of arg0
write 3 to field 'length' of arg0
return 'a'
```

The trace contains reads from and writes to the array parameter `arg0`, with writes showing what value was written but not *how* the value was computed.

Given traces based on the initial inputs, Mimic generates other potentially interesting inputs whose traces may clarify the behavior of the function. In the above trace, we see entry `read field 1 of arg0`, which reads `'b'` from the input array, and then `write 'b' to field 0 of arg0`, which writes `'b'`. Based solely on these trace entries, one cannot tell if `'b'` is being copied from the input array, or whether the write obtains `'b'` from some other computation. Hence, Mimic generates an input with a different value in `arg0[1]`, to attempt to distinguish these two cases. Input generation is based on heuristics, and is discussed more fully in Section 3.2.

Loops. Given a completed set of traces, our technique next tries to discover possible looping structures in the code. We first abstract each trace to a *trace skeleton*, which contains the operations performed by a trace but elides specific values and field offsets. A trace skeleton exposes the structure of the computation in a manner less dependent on the particular input values of a trace. For the above trace, the skeleton of the first four entries is `read;read;read;write;`.

Given a set of trace skeletons, Mimic then proposes loop structures by searching for repeated patterns in the skeletons, rating loop structures by how well they match (see Section 3.3 for details). Each loop structure is assigned a probability proportional to its rating, and then Mimic randomly selects a structure to use based on these probabilities. The full space of loop structures supported by Mimic is covered by runs of the tool with different random seeds.

For `shift`, the highest-rated loop structure is also the correct one, and is as follows (shown over the skeleton in a regular language-like syntax):

```
read;read;(has;(read;write;|delete;))*delete;
write;
```

The trace event `has` is a check if a given index is part of the array (which may return false for sparse arrays).

Categorization. After a loop structure is chosen, Mimic groups the traces into distinct *categories* based on their skeletons. For `shift`, two categories are created, one for traces matching the loop structure above, and one for the trace corresponding to the empty array (which does not match the structure). Mimic synthesizes models for each category separately, as described next, and then merges them to discover the final model which works for all inputs.

Search. To begin a random search for a model, we first generate a program to closely match one of the given input traces. For the trace above and the given loop structure, we generate the following initial program:

```
var n0 = arg0.length
var n1 = arg0[0]
for (var i = 0; i < 0; i += 1) {
  var n2 = 1 in arg0
  if (true) {
    var n3 = arg0[1]
    arg0[0] = 'b'
  } else {
    delete arg0[2]
  }
}
delete arg0[3]
arg0.length = 3
return 'a'
```

```
s  ::=  v := e[e]  |  e[e] := e  |  v := e(e, ..., e)
     |  if (e) {s} else {s}  |  v := e
     |  for (s ; e ; s) {s}  |  s ; s
e  ::=  ...  |  -1  |  0  |  1  | ...  |  true  |  false
     |  e + e  |  e - e  |  e < e  |  e == e  |  ! e
     |  (v, ..., v) => e
```

Figure 3: Syntax of a small object-oriented language.

This program is clearly too specific to the given trace (it uses specific values from its input array), and its loop and if conditions are incorrect.[1]

A random mutation is then applied to this program. For instance, the second line might be replaced with the statement `var n1 = arg0[n0+1]`. The mutations allow the program gradually be transformed towards a solution. To this end, the mutated program is compared with the current program in terms of how closely is matches the available traces. If it "ranks" better—as per a fitness function—the mutated program becomes the current candidate. If not (like in this case, where the mutation is worse), it still becomes the current candidate with a certain probability. This is a random search inspired by MCMC (see Section 3.4).

Our random search is able to gradually evolve this program into the code in lines 4–16 of Figure 2, which works for all non-empty arrays. Key to the efficacy of the search is a fitness function for programs that rewards incremental progress toward a correct solution.

For the category corresponding to empty arrays, the search discovers the following program:

```
var n0 = arg0.length
arg0.length = 0
// program implicitly returns undefined
```

Merging. After generating models for all categories, Mimic merges the models using conditional statements to create a single model. At first, the branching conditions are unknown, so some trivial condition such as `true` is used. Another phase of random search discovers the correct conditions to make the model work for all input traces. Finally, a last cleanup search makes the program more readable by removing unnecessary statements and simplifying the program. For `shift`, this final search yields the model in Figure 2.

3. APPROACH

In this section we detail the different phases of our approach to synthesizing models of opaque code, shown in Figure 1. As input, we require a function and one or more inputs to that function. Furthermore, we assume that we have a means of *executing* the function on an input of our choice and *recording a trace* of this execution. We talk more about how these assumptions are fulfilled for JavaScript in the setting of Mimic in Section 4.

We explain our approach for a simple, dynamic, object-oriented language with object allocation, field reads and writes, functions ("callable" objects), integers, arithmetic operations, conditionals, and loops. For simplicity, all field names are integers, and objects can be viewed as arrays, where the ith array element is stored in field $i-1$. The syntax for statements and expressions is given in Figure 3.

A program trace captures the execution of a function on a particular input and is a sequence of trace *events* and can be field reads, writes and function invocations. Finally, the trace contains a return value. For all the values in the trace (such as the field being read, or the receiver), the trace directly holds that value if it is a primitive (i.e., an integer). If the value is an object, then the trace holds a unique identifier for that object. For our simple language, this trace format is sufficient to capture all object accesses performed by opaque code to objects passed as parameters. Note that the trace does not contain information about where values originate.

3.1 Main Loop

Algorithm 1 Main loop for model synthesis

Require: opaque function f, initial inputs I, timeout t

```
1: procedure FindModel(f, I, t)
2:     m ← null
3:     while m = null do
4:         InitRandomSeed()
5:         m ← MimicCore(f, I) run with timeout t
6:     return m
```

Algorithm 1 gives pseudocode for the main loop of our technique. In each iteration, a global pseudo-random number generator is initialized with a fresh random seed, and then the synthesis procedure MimicCore (shown in Figure 1) is invoked with some timeout t (*null* is returned if t is exceeded). Recall from Figure 1 that the loop detection and search phases make use of randomness. Hence, by periodically restarting the search with a fresh random seed, we ensure that (1) different loop structures are explored and (2) the core search explores different parts of the state space, breaking out of local minima. Note that the loop parallelizes trivially, by simply running multiple iterations concurrently and stopping when any iteration succeeds. In our implementation, we used an exponential backoff strategy to vary the timeout (see Section 5).

3.2 Input Generation

In order to generate an accurate opaque function model in our approach, we require a set of inputs that covers all of the possible behaviors the function can exhibit. While generating such an input set is impossible in general, we have developed heuristics that have worked well for the opaque functions we have tested thus far. Given one or more representative inputs provided by the user, we automatically generate additional inputs, with three main goals:

1. Determine the flow of values through a method. As illustrated with the example trace in Section 2, with only a trace it is impossible to know if a method copies a value from another location or computes it from scratch.
2. Find dependencies on values of the inputs. For instance, the method `Array.prototype.indexOf` in JavaScript returns the index of a value in an array (or `-1` if it doesn't exist in the array). By changing the values in the array and what value is searched for, we can learn this dependency on the input values.
3. Expose corner cases. For many functions it is possible to increase the coverage by trying corner cases such

[1] In our implementation, the initial program is actually slightly more general and contains an auxiliary result variable as well as other ways to break out of the loop; see Section 3.

as the empty array. For instance, this reveals that `Array.prototype.pop` returns `undefined` for the empty array.

Algorithm 2 Input generation

Require: initial inputs *init*

procedure InputGen(*init*)
  $R \leftarrow init, I \leftarrow init, I' \leftarrow \emptyset$
  while $I \neq \emptyset$ **do**
    for $i \in I$ **do**
      $t \leftarrow$ GetTrace(i)
      $L \leftarrow$ ExtractReadLocs(t)
      $I' \leftarrow I' \cup$ GenNewVals(i, L)
    $I \leftarrow I' - R, R \leftarrow R \cup I, I' \leftarrow \emptyset$
  return R

Algorithm 2 gives pseudocode for input generation. In each iteration of the **while** loop, new inputs I' are generated from a set of inputs I as follows. For every input i in I, the opaque function is first executed on i to record a trace t. From t, we extract all memory locations L that were read, where a memory location consists of a base object and field offset. We generate new inputs by calling GenNewVals, which heuristically replaces the values of input locations in L with new values.

To decide what different values to generate for a location, we use heuristics based on the type of the original value. For integers, we randomly choose values, and include a few fixed values that are likely corner cases such as `0`, `-1`, or `1`. For objects, we sometimes replace the object with a clone, to distinguish cases involving aliased parameters. Furthermore, we can remove some fields, add additional fields, or provide an empty object.

At the outermost level, we repeat the steps above until no new inputs can be generated. In our prototype, we terminated input generation after two iterations of the **while** loop, as we found that this generated a sufficient set of inputs.

3.3 Loop Detection

After input generation, our technique next discovers possible control-flow structure for the function, i.e., loops and conditionals (the "Loop Detection" and "Categorizer" phases in Figure 1). By fixing the control flow structure for the model first, the core random search can concentrate on finding the correct expressions to complete the program. Here we describe loop detection in more detail; categorizing and merging were described in Section 2.

Given a set of execution traces like ours, discovering arbitrary looping constructs (with nesting, complex conditionals within loops, etc.) is a highly non-trivial problem. The problem can be viewed as learning a regular language form only positive examples, where the examples are the program traces, and the regular language represents the control flow. Gold [5] showed that the class of regular languages is not learnable from only positive examples. However, we have had success in discovering basic looping structures with a simple, probabilistic technique based on detecting repeated patterns that appear in multiple execution traces for a function, as described below.

We restrict the methods to have a single loop, with potentially one conditional statement in the body. To discover loop-like patterns of this form in traces, we first abstract

Algorithm 3 Loop detection

Require: set of traces T

procedure LoopDetect(T)
  $S \leftarrow$ GetSkeletons(T), $C \leftarrow \emptyset$
  for $s \in S$ **do**
    $C \leftarrow C \cup$ LoopCandidates(s)
  for $c \in C$ **do**
    $score[c] \leftarrow$ Rank(c, S)
  $L \leftarrow$ Sort($C, score$)
  return $L[i]$ with probability $\alpha_{\text{loop}} \cdot \alpha \cdot (1-\alpha)^{i-1}$

the traces to *trace skeletons*, which only consist of the event kinds in the trace. Such abstraction is useful since there is often some variation in the events generated by a statement in a loop (e.g., the index being accessed by an array read). Given such a trace skeleton, we enumerate candidate control flow structures in the skeleton by simply enumerating all possible loop start points, as well as branch lengths inside the loop.[2] Consider the following trace skeleton:

```
read; read; write; call; read; write; call;
read; read; write; call; read; write; call;
```

For this example, the following control flow candidates are generated, among others (in a regular expression-like syntax):

```
(read; | write; call)*
(read; (write; call;|))*
read; (read; (write; call;|))*
```

Note that the candidates must match the full trace, otherwise the candidate could not be the control flow that generated the given trace. To avoid guessing unlikely loops, one can restrict the search to loops that have at least some number of iterations, say 3.

We repeat the procedure above across skeletons of all traces to create a complete set of loop candidates. Depending on the length and the number of traces, this set can be quite large. Our next step is to rank the loop candidates based on their likelihood. First, we can check how many traces can be explained by a given loop structure by matching it against all trace skeletons. Intuitively, the more traces that can be explained by a possible loop, the more likely that loop is correct. Secondly, if several loops match the same number of traces, then we further rank them by the number of statements in the control flow candidate, with fewer statements ranking higher.

Once the loops are sorted according to their rank, we choose one loop at random with which to continue the search, where the ith loop is picked with probability $\alpha_{\text{loop}} \cdot \alpha \cdot (1-\alpha)^{i-1}$ for some parameters $\alpha, \alpha_{\text{loop}} \in (0,1)$. α_{loop} controls the probability that no loop candidate is chosen. In our setting, we found $\alpha = 0.7$ and $\alpha_{\text{loop}} = 0.9$ to work well.

3.4 Random Search

We now describe the random search procedure at the heart of our technique (the "Search" boxes in Figure 1). We view the problem of finding a suitable code model for a set of inputs as an optimization problem, for a fitness function that

[2]This algorithm runs in polynomial time, but could get slow for very long traces, in which case optimizations might be necessary. In the cases we have looked at, this has not been a problem.

Algorithm 4 Random search for a model

Require: set of inputs I, loop structure l

procedure SEARCH(I, l)
  $T_o \leftarrow \{i \mapsto \text{GETTRACE}(i)\}$ for $i \in I$
  $m \leftarrow \text{INITIALMODEL}(T_o[i], l)$ for some $i \in I$
  $c \leftarrow \text{FITNESS}(m, T_o)$
  while $c > 0$ **do**
    $m' \leftarrow \text{MUTATE}(m)$
    $c' \leftarrow \text{FITNESS}(m', T_o)$
    if $c' < c$ **then**
      $m \leftarrow m'$
    else
      $m \leftarrow m'$ with probability
        $\min\left(1, \exp\left(-\beta \cdot (c' - c) - \gamma\right)\right)$
  return m
procedure FITNESS(m, T_o)
  $s \leftarrow 0$
  for $(i, t) \in T_o$ **do**
    $t' \leftarrow \text{GETMODELTRACE}(m, i)$
    $s \leftarrow s + \text{COMPARE}(t, t')$
  return s

evaluates a model by comparing its execution traces to those of the underlying opaque function. Finding a model then corresponds to finding a minimum for the fitness function in the highly irregular and high-dimensional space of all models. To find a model, we employ a technique inspired by Markov Chain Monte Carlo (MCMC) sampling and the Metropolis–Hastings algorithm [1], which has been recently used successfully in the domain of superoptimization [16]. The advantage of this technique is that the sampling frequency is proportional to the value of the "density" function at that point, so the search is carried out more productively.

Algorithm 4 gives pseudocode for our random search. We maintain a current candidate model m, and each iteration creates a mutated model m'. m' is then evaluated using the fitness function, and replaces the current model if it has a lower cost. Otherwise, the new model might still be accepted, with a probability that is proportional to the difference between the two costs. This allows the search to recover from local minima and makes it robust against the irregular nature of the search space. More precisely, if the new model has a higher cost c' compared to the current cost c, then the it is accepted with probability

$$\min\left(1, \exp\left(-\beta \cdot (c' - c) - \gamma\right)\right)$$

where β and γ are parameters that controls how often models that don't improve the score are accepted.[3] Empirically, we found a value of 8 for both β and γ to work well in our setting.

3.4.1 Initial Program

We generate an initial program by exactly matching one execution trace t, respecting the given loop structure (if any). So, the initial program uses the exact concrete values from t in its statements as needed. If there is ambiguity because of aliasing (e.g., the same object is passed as two parameters), we can pick any of the choices. For statements inside a loop, where values might change with every iteration, we simply use values from the first iteration. It is then the job of the random search to generalize this very specific initial program to other inputs, and generalize statements inside loops to work for all iterations.

More precisely, a trace event `read field f of o` in t is translated to the statement `n := o[f];` for a fresh local variable `n`. The trace event `write v to field f of o` is translated to `o[f] := v;` and a call `call f with a0 .. an` yields the statement `f(a0, ..., an);`. If there is a loop, then a loop header `for (i = 0; i < 0; i++)` is generated, for a fresh local variable `i`. Initially, the loop body is never executed, and it is up to the random search to find the correct termination statement. To allow breaking out of the loop early, we add an additional statement to the end of the loop body of the form `if (false) break;`.[4]

Finally, we introduce a local variable `result` for the result of the function, which gets returned at the end. To allow incrementally building the result, we additionally add the statement `if (true) result = result;` inside the loop body as well as the `if` statement containing the `break`. Initially, these are nops, but allow the search to accumulate the result if necessary.

3.4.2 Fitness Function

Our fitness function FITNESS in Algorithm 4 computes the fitness of a candidate model by comparing its execution traces on the inputs against T_o, the traces from the opaque function. To do so, it uses a function COMPARE that takes as input a trace t from the opaque function called the *reference trace*, and t' from the current model called the *candidate trace*, both generated from the same input. It then computes a *score* that measures how close the t' is to t. A score of zero indicates $t = t'$, while any larger value indicates a difference in the traces.

For our approach to work effectively, it is crucial that the random search can make incremental progress toward a good model. This incremental progress is enabled by the fitness function reflecting partial correctness of a model in the score. To this end, COMPARE gives fine-grained partial credit, even for partially-correct individual events. For instance, if the reference trace contains the event `read field 1 of #1`, and the candidate trace contains `read field 2 of #1`, then the score is lower (better) than in the case where no read event was present at all, or where the read event also read from the wrong object.

Formally, $\text{COMPARE}(t, t')$ is defined as:

$$\frac{1}{2}\left(\sum_{\substack{\text{Event}\\ \text{kind } k}} \sum_{\substack{i<|\text{evs}(t,k)|\\ i<|\text{evs}(t',k)|}} \text{dist}(\text{evs}(t,k)[i], \text{evs}(t',k)[i])\right) \quad (1)$$

$$+\left(\sum_{\substack{\text{Event}\\ \text{kind } k}} \big||\text{evs}(t,k)| - |\text{evs}(t',k)|\big|\right) \quad (2)$$

$$+\,\mathbb{1}\{\text{return-value}(t) \neq \text{return-value}(t')\} \quad (3)$$

We use $\text{evs}(t, k)$ to refer to all events in a trace t of kind k (e.g., all field reads) as a list, which can be indexed using

[3] In an approach more closely resembeling MCMC, γ would be zero. However, we found a non-zero value for γ to reduce convergence times.

[4] Alternatively, we could allow more complicated termination expression in the loop header.

the notation $\ell[i]$, and $|\cdot|$ returns the length of a list (or the absolute value, depending on context). return-value$(\cdot)$ refers to the return value of a trace, and dist$(\cdot,\cdot)$ measures the similarity of two trace events (of the same kind). It is defined as follows:

$$\text{dist}(e_1, e_2) = \frac{1}{|\text{vals}(e_1)|} \sum_{i<|\text{vals}(e_1)|} \mathbb{1}\{\text{vals}(e_1)[i] \neq \text{vals}(e_2)[i]\}$$

where vals$(\cdot)$ is a list of values for a given trace event. For a field read, this is the receiver and field name; for a field write it additionally includes the value written; and, for a function call it includes the function invoked as well as all arguments. Note that dist$(\cdot,\cdot)$ is scaled to return a value between 0 and 1.

In the formula for COMPARE, (1) calculates the difference for all the events that are present in both traces, while (2) penalizes the model for any event that is missing or added (compared to the reference trace). Because (1) is scaled by $\frac{1}{2}$, the search gets partial credit for just having the right number of events of a given kind in a trace, even if the wrong fields are read/written. This is useful to determine the correct loop termination conditions without necessarily having a correct loop body already. Finally, (3) ensures that the result is correct. Section 5 will show the advantage of this fine-grained fitness function over a more straightforward approach.

3.4.3 Random Program Mutation

The control flow structure is fixed at this point, and the random search can concentrate on finding the correct sub-expressions for all the statements. To this end, a statement is selected at random, and modified randomly. For field reads, writes and function calls, a sub-expression is selected at random and replaced with a random new expression (we will explain shortly what kinds of expressions are generated). Similarly, for local variable assignments, the right-hand side is randomly replaced. For a `for` loop, the upper bound is replaced with a new random expression. For an `if` statement, the condition is replaced randomly. `break` statements are not modified.

Generating random expression follows the grammar of the programming language in Figure 3. To avoid creating very large nested expressions, the probability of producing an expression of depth d decreases exponentially with d. The set of local variables than can be generated is determined by the set of variables that is defined at the given program point. The set of constants is taken from all constants that appear in any of the traces, as well as a few heuristically useful constants (such as `0`).

3.5 Cleanup

When the random search succeeds, the resulting program typically contains redundant parts. For instance, if it is not necessary to break out of a loop early, then the statement `if (false) break;` can be removed. Similarly, there might be unused local variables, or expressions that are more complicated then necessary (e.g., `1+1`). To clean up the resulting model, another random search is performed, with a slightly modified fitness function as well as different random mutations: In addition to the existing program transformations, statements can now also be removed. Furthermore, we add the number of AST nodes of the program to the cost. This allows the search to improve the program by removing unused statements and simplifying expressions like `1+1` to `2`. Furthermore, the cost will never reach zero, and thus the search is stopped after a certain number of iterations have been carried out. Cleanup is not strictly necessary, as the models perform the same observable behavior, whether they are cleaned up or not. However, cleanup can make programs nicer to look at by humans.

4. IMPLEMENTATION

We have implemented the ideas presented in Section 3 in a prototype implementation called MIMIC for JavaScript. The tool is open source and available online.[5] In this section we highlight challenges and solutions particular to our setting, and discuss some implementation choices.

Trace Recording using Proxies. We leverage the ECMA-Script 6 proxy feature [13], which allows for objects with programmer-defined behavior on interactions with that object (such as field reads and writes, enumeration, etc.). An object `o` can be virtualized with a handler `h` as follows:

```
var p = new Proxy(o, h);
```

Any interaction with `p` will ask the handler how to handle that interaction (and default to accessing `o` directly if the handler does not specify any action). We leverage this feature to record the traces required for our technique, by proxying all non-primitive arguments to the opaque function. This way, we can intercept any interaction the opaque function has with the arguments and record it as an event in the trace. Our handler responds to all interactions just like the underlying object would (by forwarding all calls), with the additional book-keeping to record the trace.

Newly Allocated Objects. One challenge with this approach is that newly allocated objects will only be visible to the recording infrastructure when they are returned (or otherwise used, e.g., written to a field). For instance, the function `Array.prototype.map` takes an array and a function, and applies the function to all arguments, returning a new array with all the results. When we record the trace, we see all the accesses to the array elements and the function invocations, but the field writes to the newly allocated array are not visible.

To handle such functions, we generate the relevant object allocation at the beginning of the initial model and assign it to the `result` variable. We also add a number of guarded field writes to the newly allocated object at different locations in the model (e.g., before the loop or inside the loop body): `if (false) result[0] = 0;`. The random search is then able to identify which particular assignments are correct (by changing the guard) and find the correct field name and value to be written.

Non-Terminating Models. It is easy for the random mutations to generate programs that are non-terminating, or take very long to terminate. This can cause an infinite loop when recording the traces as part of evaluating the fitness function. Luckily, given the reference trace, we know how long the trace from the model should be, and we can abort the execution of a model when the trace gets significantly longer than the reference trace. In that case, the fitness

[5] https://github.com/Samsung/mimic

function assigns a very large cost to the trace (note that small increases in trace length are not excessively penalized).

JavaScript-Specific Features. JavaScript contains various features not in our language from Section 3. Many of these are straightforward to support, e.g., deletion of fields, function receivers, and exceptions. JavaScript allows variadic functions by providing the special variable `arguments` that can be used to access any argument using `argument[i]`, as well as the number of arguments passed with `arguments.length`. For instance, the function `Array.prototype.push` adds zero or more elements to an array. MIMIC supports such functions, by generating inputs of different lengths and generating expressions of the form `arguments[i]` and `arguments.length` during the random search. Essentially, we can view the function as just having a single argument that is an array, and then use the usual input generation strategy described earlier. The only difference is that we do not get any information in the trace about accesses to `arguments` (this special object cannot be proxied).

Optimizations. It is important for the random search to be able to evaluate many random proposals, which is why we implemented the following optimizations to our approach.

Input Selection. Input generation can create thousands of inputs (or more), and executing all of them during search would be prohibitively slow. For this reason, we restrict the inputs used during search to a much smaller number. We found 20 inputs to work well in our experiments, if the inputs are diverse enough. To get diverse inputs, we choose input that generate traces of different lengths, as well as inputs that have different scores on the initial program. If the search succeeds, all inputs are used to validate the result. If in this final validation, the score for some input is non-zero, then the search is considered to have failed, and a new run (with a different) random seed as described earlier is necessary. We found this to not be an issue.

Program Mutations. If the program mutations are generated naïvely, it is easy to generate nonsensical programs. For instance, by just following the programming language grammar, one might generate an expression that tries to read a field of an integer, or use an array object as a field offset. To avoid exploring such malformed programs, we filter out expressions when we can statically decide that they are invalid. Given the dynamic nature of JavaScript, this is not always possible, but we found that our filtering eliminates many common malformed expressions.

Parallelization Strategy. Our procedure FINDMODEL is embarrassingly parallel. However, one can further improve performance by exploiting the fact that some of the successful runs finish much more quickly than others. It is often better to kill a search early and retry again, in the hope of finding one of those very quick runs. To do this, we start with a small initial timeout t_0, and then exponentially increase it by a factor f. We found that running 28 threads in parallel with a timeout of $t_0 = 3$ seconds to work well, with a factor of $f = 2$.[6] All threads run with the same timeout, and if any of them succeed, all others can be aborted and the model can be returned. If none succeed, then the timeout is increased by a factor of f, and the process starts again.

Cleanup As noted earlier, cleanup is not necessary, strictly speaking, as models exhibit the same observable behavior with or without cleanup. For this reason, our prototype implements a quick cleanup that only removes unnecessary statements and is always applied. Then, a full cleanup as a random search can be applied optionally, to make the models more readable.

5. EVALUATION

Here, we present an experimental evaluation of MIMIC on a suite of JavaScript array-manipulating routines. We evaluated MIMIC according to the following five research questions:

(RQ1) Success Rate: How often was MIMIC successful in generating an accurate model for the tested routines?
(RQ2) Performance: When it succeeded, how long did MIMIC take to generate a model?
(RQ3) Usefulness: Were the models generated by MIMIC useful for a program analysis client?
(RQ4) Obfuscation: Is MIMIC robust to obfuscation?
(RQ5) Fitness Function: How important was the fine-grained partial credit given by our fitness function (Section 3.4.2)?

5.1 Experimental Setup

Our primary subject programs were the built-in methods for arrays on `Array.prototype` provided by any standard JavaScript runtime [12]. These methods exhibit a variety of behaviors, including mutation of arrays, loops that iterate forward and backward through the array, and methods returning non-trivial computed results (e.g., `reduce`). Furthermore, many of these methods can operate on JavaScript's sparse arrays (in which certain array indices may be missing), necessitating additional conditional logic.

We ran our experiments on a Intel Xeon E5-2697 machine with 28 physical cores, running at 2.6 GHz and using `node v0.12.0` to execute MIMIC (written in TypeScript and Python). Our implementation parallelizes the search for a model using different random seeds.

5.2 Results

Success Rate MIMIC was able to generate models for some of the `Array.prototype` functions, but not others. The functions for which it succeeded are listed in Table 1; we discuss the other functions below. The models we synthesized can be found at https://github.com/Samsung/mimic/blob/master/models/array.js.[7] We also included three other functions over arrays (`max`, `min`, and `sum`) that performed a bit more computation than the built-in functions.[8]

Performance Experimental data addressing **(RQ2)** appears in Table 1. The performance of MIMIC was quite reasonable, with models taking an overall average of 60.6 seconds to generate using our exponential increasing timeout strategy, and less than 5 minutes on average for all models. This experiment was repeated 100 times per example, without using the full cleanup (see Section 4). The additional time it would require to perform a full cleanup is 3.74 seconds

[6] If run with fewer threads, MIMIC will automatically use a smaller factor, so that roughly the same number of short runs are made.

[7] Some array methods can handle "Array-like" objects; we have not used such inputs and focused only on actual arrays.

[8] Though code is available for these functions, MIMIC made no use of it, observing their behavior as if they were opaque.

Table 1: Summary of all `Array.prototype` functions that Mimic can handle, as well as some handwritten functions to compute the sum, maximum and minimum of an array of numbers. We report the average time (over 100 runs) it takes to synthesize the model (using the quick cleanup) on our hardware as well as how high the correct loop was ranked (1 being the highest rank).

Function	Time to synthesize (in seconds)	Loop rank
every	67.86 ± 22.41	1
filter	43.05 ± 16.13	1
forEach	4.59 ± 1.52	1
indexOf	42.94 ± 38.59	1
lastIndexOf	36.92 ± 19.22	2
map	15.64 ± 6.86	1
pop	2.35 ± 0.74	loop-free
push	291.94 ± 310.17	3
reduce	25.33 ± 12.51	1
reduceRight	126.41 ± 53.77	1
shift	117.54 ± 52.11	1
some	6.74 ± 3.16	1
max	56.61 ± 107.86	2
min	50.69 ± 121.95	2
sum	20.35 ± 39.83	2

on average, and less than 8 seconds for all examples (for the default of 1000 cleanup iterations).

In the same table we also show the loop rank assigned by our ranking heuristic (1 being the highest ranked proposal). The loop ranking chooses the correct control structure for 9 out of 15 examples with rank 1, and with our choices for α and α_{loop}, MimicCore then picks this loop with probability 64%. For the five examples where the correct loop is ranked second, the probability is 18.9%, and for `push` with rank 3 it is 5.7%. Finally, `pop` does not have a loop (and the loop inference does not propose any loops).

Longer search times were due to several reasons: (1) loop ranking, (2) complicated expressions (e.g., generating the index `n0-i-1` is lower probability than, say, `i`), and complex dependencies between loop iterations (`reduce` requires the result of the one iteration to be passed in the next iteration).

Mimic currently cannot synthesize models for `Array.prototype` methods not listed in Table 1, due to the following issues:

- **Multiple loops**: `concat`, `sort`, `splice` and `unshift` all require multiple loops, for which we currently do not have any heuristics.
- **Complex conditionals within loops:** `reverse` reverses a list in place. The loop body takes two indices from the front and back and exchanges them. To handle sparse arrays, four different cases need to be handled (depending on whether either element is present or not).
- **Lack of relevant mutations:** `join`, `toString` and `toLocalString` require computing a (possibly localized) string representation of an arbitrary array element, which our mutations do not propose at the moment.
- **Proxy-related bugs:** We discovered some bugs in the current proxy implementation in `node.js`. Unfortunately, it crashes for `slice`. It also reports traces that are not in accordance with the specification for `concat`, `shift` and `reverse`. For `shift` we used our own implementation that follows the specification.

Usefulness To answer **(RQ3)**, we compared the models generated by Mimic with those present in the T.J. Watson Libraries for Analysis (WALA) [20], a production-quality static analysis framework for Java and JavaScript. We found that WALA did not yet have any model for functions `reduce`, `every`, and `some`. Since these functions perform callbacks of user-provided functions, a lack of a model could lead to incorrect call graphs for programs invoking the functions. We added the Mimic-generated models for these functions to WALA, and confirmed via small examples that their presence improved call graph completeness. Additionally, we found WALA's existing model of the frequently-used `pop` function was written incorrectly, such that it would always return `undefined` rather than the popped value. Many of WALA's models also do not handle sparse arrays, which Mimic models do handle. These examples illustrate how writing models by hand can be tedious and error-prone, and how tool support can ease the process. Mimic-generated models for the above functions were accepted for inclusion in WALA.[9]

Obfuscation To answer **(RQ4)**, we ran Mimic-generated models through a well-known JavaScript obfuscator,[10] and tested that Mimic generated the same model for the obfuscated function. The obfuscator employed rewrites static property accesses into dynamic ones (see Section 1), which could significantly degrade the quality of many static analyses. We confirmed that for all functions, we obtained the same model when using the obfuscated code as the baseline, showing the promise of our technique for making such code more analyzeable.

Fitness Function To answer **(RQ5)**, we ran Mimic with a more naïve fitness function that only gave credit to a trace when an individual event matched exactly with the reference trace, rather than giving partial credit for partial event matches (see Section 3.4.2). In an experiment, we found that our fitness function led to decreased running times compared to the naïve function, validating our use of fine-grained partial credit. Specifically, the running time average increased by 39.7% on average for the naïve fitness function, and as much as 170% for some examples.

5.3 Limitations and Threats to Validity

Mimic currently cannot handle opaque functions with any of the following properties:

- *Complex local computations*, e.g., a sine function, as our trace does not expose such computations.
- *Nested loops or complex conditionals inside loops*, as discussed above. Adding such support without excessively increasing the number of loop candidates is future work.
- *Reliance on non-parameter inputs*, e.g., a variable in the closure state initialized via some other sequence of method calls. This could perhaps be handled by the user providing a richer test harness for the function.
- *Global side effects*, e.g., on the network or file system. This is a limitation of the native execution approach.
- *Long running time*, which will slow down our search, again due to native execution.

[9] https://github.com/wala/WALA/pull/64

[10] http://www.javascriptobfuscator.com/Javascript-Obfuscator.aspx

The primary threat to validity of our evaluation is our choice of benchmarks. We tested MIMIC on all `Array.prototype` methods, and discussed cases it could and could not handle. But, MIMIC's handling of different array routines, or routines on other data structures, could vary. Our results thus far are quite encouraging, and we plan to further validate MIMIC for other types of functions in future work.

6. RELATED WORK

Summary Computation Computing summaries of procedure calls has been a very active area of in program analysis, and it is impossible to mention all the related work on the topic; Sharir and Pneuli's seminal work [17] is a good overview of the foundational approaches to this problem. We have used the term "models" in this paper; summaries can be thought of as models that suffice for specific static analyses, which use abstractions.

Most of the work on summary computation assumes the availability of library code for analysis. However, recently, several authors have tried to deal with unavailable libraries. The work by Zhu et al. [21] deals with the problem of missing library code by inferring the minimal number of "must not flow" specifications that would prevent undesirable flows in the client application. Bastani et al. [2] have also presented an approach that infers potential summaries for an unavailable library, based on a particular static analysis problem being solved. In both the above approaches, ultimately a human auditor must judge the validity of the findings of the tool. Like these works, we also do not require the code to be available in analyzable form, but (unlike these works) we do assume the library code is available to execute. On the other hand, while the summaries computed in the above mostly capture tainted-ness and aliasing, the models that we compute are considerably richer.

Madsen et al. [11] construct models of library functions through static analysis of uses of values returned from native JavaScript functions. However, their work does not infer flow of values *through* the library functions as would be needed, for instance, for alias analysis; rather, it only infers the type structure of values returned from the libraries.

Summaries can be helpful in dynamic analysis settings as well. Palepu et al. [14] compute summaries of library functions with the intention of avoiding repeated execution of instrumented library code. Their summary representation is suitable for aliasing and taint-flow like properties. By contrast, our technique requires only partial observation of the execution of the library code, and our models are richer, but they are more expensive to compute.

Trace-based Synthesis As mentioned in Section 1, there has been considerable work related to the problem of constructing programs to fit traces. One of the earlier papers in this area is that of Biermann et al. [3]. Given a trace of *all* instructions, as well as conditionals, they show how to construct a (looping) program that would explain the trace. They present a backtracking procedure that tries to discover the right program structure. However, the technique requires all instructions, including conditionals, in the trace. By contrast, our technique requires observing only the shared reads and writes.

Traces can be detailed traces of low-level instructions, or they can be traces of high level domain-specific instructions, e.g. deleting a character in an editing buffer. Automatic construction of "programs" to automate end-user tasks has been an area of much work in the last decade [10, 6]. Repetitive user actions correspond to example traces, and a tool synthesizes a program representation that automates such user actions. However, most current approaches to synthesis from examples are limited to generating programs in carefully-designed domain-specific languages, leveraging clever algorithms and data structures for efficiency (cf. version algebra discussion in Section 1). Extending such approaches to generating programs with richer data and control structures, as needed in our scenario, is non-trivial. When it comes to general purpose programs, Lau et al.'s work [10] mentions the increased difficulty in synthesis when traces do not capture all the steps of a program's execution; in fact, they report experiments only on complete traces, as in Biermann's work.

Other Synthesis Approaches Jha et al. [8] presented a technique to synthesize straight-line programs from examples. Their technique searches through a space of combinations of program operations which could be either primitive statements, or API calls. Like our work, they check the correctness of their synthesized programs against a native implementation on a set of inputs. Later, Qi et al. [15] extended the technique to handle loops and data structures by extending the set of possible primitive statements. As discussed in Section 1, the main limitation of this approach is in scalability of the constraint solver as the number of primitive statement types increases; our generate-and-test approach leverages native execution and parallelization to improve scalability.

Sketching [18] is another synthesis system based on constraint solving. The idea of sketching is to synthesize missing expressions (a.k.a. "holes") from an otherwise complete program. For certain domains, boolean satisfiability (SAT) techniques can be used to efficiently synthesize completions of programs with holes. Kuncak et al. [9] have presented automated synthesis procedures that work (based on formula solving) for a limited setting of linear arithmetic and data structures. In our setting, the entire program needs to be synthesized, and hence the above constraint-based techniques do not apply directly.

Other Uses of Search Our work is inspired by recent successes of search-based techniques in the research community. To pick two examples, there is the automatic patch generation work by Weimer at al. [4], which uses genetic algorithms for search (and which we discussed in Section 1), and compiler optimization work by Schkufza et al. [16], which also uses the Markov-Chain Monte Carlo technique.

7. CONCLUSIONS

We have presented a novel technique for synthesizing models for opaque code. We collect partial execution traces by intercepting memory accesses to shared state, and then use a random search technique to construct an executable code model that matches the traces. We implemented our technique in a tool MIMIC for JavaScript, and showed that it could synthesize non-trivial models for a variety of array-manipulating routines.

Along with the paper we have submitted a replication package containing the full source code of our implementation as well as all the experimental setup to reproduce the results in this paper. It has been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations.

8. REFERENCES

[1] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[2] O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free language reachability. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 553–566, 2015.

[3] A. W. Biermann, R. I. Baum, and F. E. Petry. Speeding up the synthesis of programs from traces. *IEEE Trans. Comput.*, 24(2):122–136, Feb. 1975.

[4] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 947–954, New York, NY, USA, 2009. ACM.

[5] E. M. Gold. Language identification in the limit. *Information and Control*, 10, 1967.

[6] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[7] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of javascript web applications. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 59–69, 2011.

[8] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224, 2010.

[9] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 316–329, New York, NY, USA, 2010. ACM.

[10] T. A. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *Proceedings of the 2nd International Conference on Knowledge Capture (K-CAP 2003), October 23-25, 2003, Sanibel Island, FL, USA*, pages 36–43, 2003.

[11] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 499–509, 2013.

[12] M. D. Network. Array.prototype. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/prototype. Accessed: 2015-03-15.

[13] M. D. Network. Proxy - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy. Accessed: 2015-03-16.

[14] V. K. Palepu, G. H. Xu, and J. A. Jones. Improving efficiency of dynamic analysis with dynamic dependence summaries. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 59–69, 2013.

[15] D. Qi, W. N. Sumner, F. Qin, M. Zheng, X. Zhang, and A. Roychoudhury. Modeling software execution environment. In *Working Conference on Reverse Engineering*, WCRE'12, pages 415–424, Washington, DC, USA, 2012. IEEE Computer Society.

[16] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In V. Sarkar and R. Bodík, editors, *ASPLOS*, pages 305–316. ACM, 2013.

[17] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, 1981.

[18] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.

[19] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: taint analysis of framework-based web applications. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 1053–1068, 2011.

[20] T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net.

[21] H. Zhu, T. Dillig, and I. Dillig. Automated inference of library specifications for source-sink property verification. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, pages 290–306, 2013.