Final Year Project Report

Full Unit - Final Report

Array Support for Dynamic Symbolic Execution of JavaScript

Arran France

A report submitted in part fulfilment of the degree of

BSc (Hons) in Software Engineering

Supervisor: Dr Johannes Kinder

Department of Computer Science
Royal Holloway, University of London

March 28, 2018

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 18,038

Student Name: Arran France

Date of Submission: 2018-03-28

Signature: Arran France

Table of Contents

Abstract

Dynamic symbolic execution is a technique that can used on JavaScript to identify bugs and generate test cases, as the recent tool ExpoSE does. JavaScript arrays are an area of the language which have atypical behaviour and as a result are challenging to model. I present an encoding of homogeneously typed JavaScript arrays in first order logic and describe extensions to the symbolic execution tool ExpoSE which implement support for symbolic arrays and a selection of Array prototype functions. I demonstrate that these extensions lead to an increase in path count and coverage on two popular JavaScript libraries and describe a technique using ExpoSE which was used to identify a bug in a popular polyfill library.

Chapter 1: **Introduction**

Since its inception as a ‘glue’ language for the web [1] JavaScript’s popularity has surged [2, 3]. JavaScript is now a ubiquitous language that powers mobile, web, and server applications however its dynamic nature and deceptive syntax often lead to subtle bugs and unintended behaviour which can be difficult to identify. Dynamic symbolic execution (DSE) is a technique that can be used to automatically identify bugs and generate tests for JavaScript however this requires modelling the semantics of the language in first order logic. JavaScript arrays are an area of the language that requires trade-offs, difficult design decisions, and careful consideration of language semantics when modelling due to their atypical properties.

My work extends ExpoSE [4], a tool for DSE, by describing an encoding for homogeneously typed JavaScript arrays in first order logic, implementing support in ExpoSE for symbolic arrays, and modelling a selection of prototype functions. I demonstrate that these extensions result in an increased test coverage of two popular JavaScript libraries and describe a new bug found in a popular Array ES6 polyfill.

I begin this report presenting background information on symbolic execution, including considerations for constraint optimisations and its usage for bug detection, automated decision procedures, with an emphasis on techniques for SAT and SMT solving, and a brief overview of JavaScript’s semantics, with a particular emphasis on the behaviour of arrays and their prototype functions. I then describe the details of the implementation of my extensions to ExpoSE, my experimental methodology, and draw conclusions about their effectiveness on testing real world applications.

Chapter 2: **Symbolic Execution**

2.1 Overview

Symbolic execution is a technique for testing program correctness initially proposed in the 1970s [5, 6]. Classically, it is the process of executing a program by representing its variables as symbols rather than concrete values. Execution of the program occurs using these symbolic inputs by extending the basic operators and semantics of the program's language to use symbols and produce symbolic formulas instead of concrete values. In doing so, each symbolic execution tests many actual instances of execution. `If` statements and language features that cause execution to diverge are represented as a path condition, a conjunction of conditions on symbols that must be satisfied. At the end of execution, the path condition, the conjunction of branch conditions taken to the current point of execution, provides a unique representation of a path tested through the program. The path condition when solved, using a constraint solver, produces a set of inputs that when executed take the same path through the program as the symbolic execution [7, 8].

For example, consider the program described in Figure 2.1 which returns the sum of the absolute values of two values. There are four possible paths that can be taken through the program: the first is where neither `if` branches are entered, the second is where the value of `x` is less than zero, the third is where the value of `y` is less than zero, and the fourth is where both `x` and `y` are less than zero. The process for symbolically executing this program classically is as follows: the function is executed with two symbolic values x' and y' and a path condition pc, "the accumulator of properties which inputs must satisfy ... to follow ... a particular path" [5], is initialised to *true*. When a branching statement (line 2 or 5) is reached the constraint solver is queried to see if the branch condition is feasible given the current path condition. If the condition is satisfiable then execution forks to explore the new path and pc becomes $pc \wedge bc$ where bc is the branch condition. This process repeats until the program terminates and every path is explored. Table 2.1 shows the path condition, symbolic results, and the constraint solver queries and results for the execution of Figure 2.1.

Historically, this technique has been impractical due to the limitations on computing resources and the capabilities of automated theorem provers [5]. However in the 2000s there were a number of advancements in the technique by DART [8] and EXE [9] which, combined with improvements in computer hardware and SMT solvers [10], made the technique practical.

In these modern implementations, the program is run both concretely and symbolically in, so called, *concolic execution*. In concolic execution, the program is instrumented to run concretely whilst the program state is shadowed by symbolic variables, allowing the concrete execution to drive the symbolic execution. Unlike pure symbolic execution, concolic execution requires its initial input to be concrete; this input can be random or crafted to fit the target program [11, 12]. As concolic execution concretely executes programs, rather than just modelling the semantics of the language, one can be sure that any potential errors found are actual bugs. Additionally, when a constraint cannot be solved, concolic execution allows concrete values to be substituted for unsolvable elements of constraints [13, 14].

Consider the following example of concolic execution testing the program described in Figure 2.1. Inputs for `x` and `y` are randomly generated, for instance $x = 217$ and $y = 931$. These inputs then guide the execution to not visit any branches, producing the pc of $x >= 0 \wedge y >= 0$. This path condition can then be negated to produce further paths. Using the DART [8] style directed symbolic execution paths are generated as per Table 2.2.

```
function abs_sum (x, y) {
 if (x < 0) {
   x = x * -1;
 }
 if (y < 0) {
   y = y * -1;
 }
 return x + y;
}
```

Figure 2.1: Absolute Sum Function.

pc	Result	Constraint Solver Queries	Query Result
$x >= \wedge y >= 0$	$x + y$	$x >= 0 \wedge y < 0$	SAT
$x >= \wedge y < 0$	$x - y$	$x < 0$	SAT
$x < \wedge y >= 0$	$-x + y$	$x < 0 \wedge y < 0$	SAT
$x < \wedge y < 0$	$-x - y$		

Table 2.1: Absolute Sum Symbolic Execution Results.

2.2 Finding and Choosing Paths

Although, in theory, symbolic execution will exhaustively search through all feasible paths, as the length of the program under test increases the number of paths in the program increases roughly exponentially [12]. As these symbolic execution tools are bounded by real world time constraints, paths must be chosen carefully in order to prioritise exploring 'interesting' cases in the time available.

In EXE, and other more traditional methods, paths are found by building a control flow graph and then exploring the graph using a form of breadth-first search (BFS) or depth-first search (DFS) algorithm. Neither BFS or DFS are particularly well suited for the problem as neither is able to prioritise 'interesting paths', they simply try to exhaustively explore all the paths in a program. DFS is especially flawed due to its ability to get stuck in a symbolically bounded loop [9].

SAGE, by contrast, uses a *generational search*. Following the initial input, paths are generated by repeatedly negating the conditions in the path condition that was explored. Then paths are prioritised using a priority queue [8].

In detail, the algorithm for finding new test inputs (Figure 2.2 and 2.3 [7]) is as follows: starting with an initial input i which has an attribute *bound* of 0 and a priority queue pq the program executes symbolically. The path condition for the path taken, pc, is then computed where pc is in the form of a sequence of predicates $c_1 \wedge c_2 \wedge ... c_n$. To explore alternatives to the branches that were chosen in the last execution new inputs are generated. For $j = i.bound...n$ where n is the number of predicates in pc, n new inputs are attempted to be generated. Inputs are generated by seeing if there is a satisfiable solution for the conjunction of $c_0...c_{j-1} \wedge \neg c_j$, if so a new input is created with the path condition of the satisfiable solution and a bound attribute of j [8, 11]. The attribute *bound* is used to prevent path conditions being explored multiple times. These new inputs are added to the the priority queue and can be prioritised using a chosen heuristic [12].

Constraint solvers are used for two purposes in symbolic execution: determining the satisfiability of branch/path constraints and producing concrete input for satisfiable conditions.

Input (x, y)	*pc*	Result	Constraint Solver Queries	Query Result
0, 0	$x >= \wedge y >= 0$	$x + y$	$x >= 0 \wedge y < 0$	SAT
0, -3	$x >= \wedge y < 0$	$x - y$	$x < 0$	SAT
-2, 1	$x < \wedge y >= 0$	$-x + y$	$x < 0 \wedge y < 0$	SAT
-2, -7	$x < \wedge y < 0$	$-x - y$		

Table 2.2: Absolute Sum Concolic Execution Results.

```
function SEARCH($i$)
    $i.bound \leftarrow 0$
    $pq \leftarrow [i]$
    SYMBOLICEXECUTE($i$)
    while $pq$ not empty do
        $inputs \leftarrow$ EXPANDINPUTS($pq[0]$)
        while $inputs$ not empty do
            $input \leftarrow inputs[0]$ SYMBOLICEXECUTE($input$)
            $pq \leftarrow pq + input$
        end while
    end while
end function
```

Figure 2.2: SAGE's Search Algorithm.

Despite continual improvements in the performance of constraint solvers, calls to solvers are typically the bottleneck of symbolic execution – fortunately there are a number of possible optimisations that can be done to reduce the burden on constraint solvers and achieve better performance [12].

2.3 Constraint Optimisations

Frequently constraints will need to be solved multiple times throughout the course of execution. Instead of calling the constraint solver multiple times for the same constraint, satisfying results can be stored in a map of constraints to satisfying results. Then, instead of calling the constraint solver directly, a cache lookup can be performed first to see if there is a satisfying result [15].

This caching process can be extended to evaluate the strength of formulas. If there is a satisfying result for a stronger constraint than the one being queried, then that satisfying result will be valid for the weak constraint. For instance, if the cache contains the following mapping $(x > 3 \wedge y < 1) \wedge (x < 10) \implies x = 5 \wedge y = -214$ then the satisfying result will be valid for the condition $x > 3 \wedge y < 1$. This caching scheme can also be used to cache unsatisfiable constraints [15].

In most cases, a branch's condition will not depend on all of the variables in a path constraint, allowing redundant constraints to be eliminated before passing the constraint to the constraint solver. Although, if this technique is applied, the constraint solver will only return satisfiable values for non-eliminated variables, the existing concrete values of the other variables can be used to produce a full set of inputs [12].

For instance, consider the function in Figure 2.4. If the existing path constraint is $(x < 0) \wedge (z > 5) \wedge (y < 13) \wedge (y + x > 15)$ and the symbolic execution negates the the last path condition to explore a new path: $(x < 0) \wedge (z > 5) \wedge (y < 13) \wedge \neg(y + x > 15)$, then the $z > 5$

```
function ExpandInputs($input$)
    $alternatives \leftarrow [\,]$
    $pc \leftarrow$ ComputeConstraint($input$)
    for $j = pc.bound...|input$ do
        if $pc[0 : j - 1] \wedge \neg pc[j]$ has a satisfiable solution $I$ then
            $newInput \leftarrow I$
            $newInput.bound \leftarrow ij$
            $alternatives \leftarrow alternatives + newInput$
        end if
    end for
    return $alternatives$
end function
```

Figure 2.3: SAGE's Expansion Algorithm for Generating New Paths.

```javascript
function unlikely_function (x, y, z){
 if (x < 0)
   x--;
 if (z > 5)
   z++;
 if (y < 13)
   y++;
 if (y + x > 15)
   y = y * x;
 return x + y + z;
}
```

Figure 2.4: Example Function.

constraint can be eliminated as z does not influence the $y + x > 15$ branch.

2.4 Bug Detection

Symbolic execution can be used for a variety of purposes: generating test inputs, finding infeasible paths, and finding equivalent (and therefore redundant) code but its typical application, and the focus of this project, is bug detection.

In classical symbolic execution, as no actual code is executed bugs, must be detected through the use of logical assertions that are tested by the constraint solver. EFFIGY, described in [5], uses the concept of an input predicate and an output predicate to determine program correctness. If all inputs which satisfy the input predicate also satisfy the output predicate then the program is said to be correct. These are implemented through the use of `ASSERT`, `ASSUME`, and `PROVE` statements. In modern symbolic execution tools, such as DART bugs, are detected either when computed properties such as memory-safety are violated or runtime exceptions occur [8], rather than through the use of additional assertions.

Symbolic execution offers a number advantages over typical testing techniques. Unit and integration testing are expensive and require a large amount of effort to reach 100% coverage. In addition, they require a large amount of mock and driver code to be written, often exceeding the size of the program under test. Unit testing and integration also frequently miss 'corner case' bugs, which are not considered either when the program is developed or when the system is tested. Conversely, symbolic execution automatically generates inputs that will cover all paths, specifically targeting corner cases which are likely to be missed by developers. Although symbolic execution may require some mocking and harness code to be written, the amount of code and effort required to write it is significantly less than writing either a unit or integration test suite.

Static analysis tools, such as Coverity [16, 17] and CodeSonar [18], are much cheaper to run but at the cost of precision. Static analysis often produces false reports and consequently also requires a large amount of time to spend identifying actual faults. Additionally, static analysis often may struggle to identify subtle bugs due to the lack of information available outside of runtime, and will typically perform worse on weakly typed languages. Symbolic execution, unlike static analysis, does require effort on the part of the developer but has the advantage of not producing false positives and providing a valid test case for each bug found. Symbolic execution is also effective at finding edge case bugs and can also effectively analyse the behaviour of a program when it interacts with libraries for which the source code is not available.

Chapter 3: **Automated Decision Procedures**

3.1 A Short History of Automated Reasoning

The notion of automated reasoning has long existed, the most prominent early example being Gottfried Leibniz's dream of a machine which could automatically determine the truth of any assertion, "If controversies were to arise, there would be no more need of dispute between two philosophers than between two accountants". Leibniz's dream was later echoed in David Hilbert's challenge to the 1928 International Congress of Mathematicians, known now as the *Entscheidungsproblem* or decision problem, in which he posed the question (paraphrased here) "Is there a mechanical procedure which when fed any mathematical proposition determines in a finite number of steps whether or not the statement is provable from a set of axioms, using first-order logic?" [19, 20].

At the time Hilbert proposed his challenge no formal notion of a mechanical procedure, or algorithm, existed. In 1935/6 Alonzo Church and Alan Turing independently defined equivalent notions of an algorithm, lambda calculus and Turing machines, and found a negative result for the *Entscheidungsproblem* by proving that first-order logic is undecidable [21].

Despite first-order logic being generally undecidable, there are some first order theories which are including Presburger Arithmetic, a first order theory of only addition over natural numbers - more limited than the general undecidable Peano's arithmetic [20]. In 1954, Martin Davis implemented a decision procedure for Presburger arithmetic producing the first computed mathematical proof. He remarked later, "Its great triumph was to prove that the sum of two even numbers is even," an acknowledgement of the achievement of the program but also its limitations [22, 20].

From the 1950s onwards there was a steady increase in the body of work related to automated reasoning including work by Davis and Putman [23, 24], John Robinson's proof of the soundness and completeness of first order resolution [25], and Simon and Newell's Logic Theorist [26]. However the interest in the field drastically increased in the 2000s following the publication of the Chaff SAT solver [27] and a renewed interested in theory solving [22].

3.2 SAT

A classic problem in the realm of automated reasoning is the SAT problem. The SAT problem asks the question "Given a formula in propositional logic is there an assignment of true/false values that makes the formula true?" [19].

For instance, given for the following formula, the answer would be no, the problem is unsatisfiable: $(a \lor b) \land (b \land \neg a) \land \neg b$.

In the 1970s, Stephen Cook and Leonid Levin showed the SAT problem was NP-complete seeming to make automated reasoning infeasible for efficient algorithms. However, although random SAT problems exhibit worst-case performance, many real-world problems tend not to due to their tendency to have structure [19].

Most SAT solvers, automated tools to produce solutions for a given SAT problem, use the Davis-Putnam-Logemann-Loveland (DPLL) backtracking search combined with Conflict-

```
function UnitPropagate($F$, $U$)
    while $F$ contains a unit clause $C$ do
        $F \leftarrow F|C$
        $U \leftarrow U \cup C$
    end while
end function
```

Figure 3.1: Unit Propagation Algorithm.

```
function PureLiteral($F$, $U$)
    while $F$ contains a pure literal $l$ do
        $C \leftarrow F|l$
        $U \leftarrow U \cup l$
    end while
end function
```

Figure 3.2: Pure Literal Algorithm.

Driven-Clause-Learning (CDCL) to determine satisfiability [28, 29].

3.2.1 DPLL

DPLL is a depth-first search complete systematic process for finding a satisfying assignment or proving unsatisfiability for a given SAT problem [24]. The algorithm, at first approximation, attempts to assign a variable a Boolean value, and checks the satisfiability of the problem (with the assigned values). If the problem is not unsatisfiable then another variable has a value assigned, otherwise the previously assigned value is negated. This process is repeated until all variables are assigned a value and either the formula is satisfiable or, no matter what variable is assigned the formula is unsatisfiable [30]. Typically SAT problems are expressed in conjunctive-normal form (CNF), as a conjunction of one or more clauses which consist of the disjunction of one or more literals.

For any CNF formula F and literal l, $F|l$ represents the formula obtained by replacing all occurrences of l in F with $true$ and simplifying by removing all clauses containing $true$ and the terms $\neg true$. DPLL relies on two main principles: simplification and case splitting. Simplification is achieved through unit propagation and pure literal deletion. If a CNF formula F contains a clause C which only contains a single literal l then the clause ‘fixes’ the value of l in all other clauses which contain l, the procedure for applying unit propagation is shown in Figure 3.1. Pure literal deletion can be applied when there exists a literal l in a CNF formula F where $\neg l \notin F$, all clauses containing the pure literal can be removed from the formula which has the effect of setting the pure literal’s assignment to true. The procedure for applying pure literal deletion is shown in Figure 3.2 [31, 32]. Case splitting occurs in the recursive call to DPLL, a truth assignment is guessed and then satisfiability is recursively checked; if the assignment is unsatisfiable then the negation of that assignment is checked effectively splitting the problem into two simpler cases.

Example Consider the problem $F = (P \vee \neg Q \vee R) \wedge (\neg P \vee R) \wedge \neg P \wedge \neg U \wedge (R \vee Q)$ and Table 3.1 which shows the result of applying the DPLL algorithm on the problem. Initially the unit clauses, $\neg U$ and $\neg P$, are propagated which reduces the problem to $(\neg Q \vee R) \wedge (R \vee Q)) \wedge (R \vee Q)$. Then the pure literal rule can be applied to R as its value is consistent across all clauses, which results in no remaining clauses and a satisfying assignment of $\neg U$, $\neg P, R$.

It is also possible to model the algorithm as a series of states and transitions [33]. States are described in the form $M||F$, where M is a sequence of literals which are assigned the value

```
function DPLL($F$, $M$)
    UnitPropagate($F$, $M$)
    if $F$ contains an empty clause then
        return unsat
    end if
    PureLiteral($F$, $M$)
    if $F = T$ then
        return sat with the model $U$
    end if
    $l \leftarrow$ a literal containing an atom from $F$
    DPLL($F|L$, $U \cup L$)
    DPLL($F|\neg L$, $U \cup \neg L$)
end function
```

Figure 3.3: DPLL Algorithm.

Model	F	Rule Applied
	$(P \vee \neg Q \vee R) \wedge (\neg P \vee R \vee Q) \wedge \neg P \wedge \neg U \wedge (R \vee Q)$	UnitPropagate on $\neg U$
$\neg U$	$(P \vee \neg Q \vee R) \wedge (\neg P \vee R \vee Q) \wedge \neg P \wedge (R \vee Q)$	UnitPropagate on $\neg P$
$\neg U$, $\neg P$	$(\neg Q \vee R) \wedge (R \vee Q)) \wedge (R \vee Q)$	PureLiteral on R
$\neg U$, $\neg P, R$	*true*	

Table 3.1: Example DPLL Results.

true and F is the problem in CNF form. The initial state of the algorithm is as $\emptyset||F$ and the final state is either a special state *unsat* if F is unsatisfiable, or $M||F'$, where F' is a CNF formula equivalent to the initial problem F and M satisfies F'. The classical algorithm can be described by the four basic transitions, $M||F \implies M'||F'$, shown in Table 3.2: *UnitPropagate*, *Decide*, *Fail*, and *Backtrack* where $M \models C$ means that for every assignment v, $v(M) = true$ implies $v(C) = true$. These rules are applied to each state until a final state is achieved.

3.2.2 CDCL

As variables are assigned Boolean values during the DPLL algorithm, conflicts arise from the implications of such assignments. For instance consider the following problem in CNF:

$(a \vee b \vee \neg c) \wedge (a \vee \neg d) \wedge (c \vee d \vee \neg f) \wedge \neg e \wedge (g \vee e \vee \neg h) \wedge (f \vee h)$

With the assignments $\neg g, \neg a, \neg b$ the problem is unsatisfiable due to the final clause, $(f \vee h)$. In order for the first clause to be true $\neg c$ must be assigned the value true, and likewise for the second clause to be true $\neg d$ must be true, and for the third clause to be made true $\neg f$ must be true. The unit clause $\neg e$ must be assigned the value true as well, which results in the model $\neg g, \neg a, \neg b, \neg c, d, \neg f, \neg e$. With this model it becomes clear that the final two clauses, $(g \vee e \vee \neg h) \wedge (f \vee h)$, cannot be assigned to to produce a satisfying result. The first clause, $(g \vee e \vee \neg h)$ requires an assignment of $\neg h$ but the last clause requires an assignment of h.

CDCL attempts to identify the reason for the conflict and then use that information to jump several decisions back in the search to try a new assignment that will avoid that conflict. This process avoids exploring search spaces where there cannot be a solution. In order to do so, additional state transitions as described in Table 3.3 are required: *Learn*, *Forget*, and *Restart*.

In DPLL with CDCL, each assignment has a *decision level* which represents the number of

Name	Rule	Conditions	Example
Unit Propagation	$M\Vert F, C \vee l \implies Ml\Vert F, C \vee l$	$M \models \neg C$, l is undefined in M	$\emptyset\Vert a \wedge (b \vee c)) \wedge (d \vee e) \implies a\Vert a \wedge (b \vee c)) \wedge (d \vee e)$
Decide	$M\Vert F \implies Ml^d\Vert F$	l or $\neg$ occurs in a clause of F	$\neg b\vert\neg b\Vert, a \wedge (\neg b \vee c) \implies \neg b \wedge c^d\Vert a \wedge (\neg b \vee c)$
Fail	$M\Vert F, C \implies unsat$	$M \models \neg C$ M contains no decision literals (l^d)	$a \wedge \neg a\Vert\neg a \wedge a \implies unsat$
Backtrack	$Ml^dN\Vert F, C \implies Ml'\Vert F, C$	$Ml^dN \models \neg C$ and there is a clause $C' \vee l'$ where $F, C \models C' \vee l'$ and $M \models \neg C'$, l' is undefined in M, and $l'or\neg l'$ occurs in F or in Ml^dN	$a^d\Vert(a \vee b) \wedge (\neg a \vee c) \implies \neg a\Vert(a \vee b) \wedge (\neg a \vee c)$
Pure Literal	$M\Vert F \implies Ml\Vert F$	l occurs in some clause of F, $\neg l$ occurs in no clause of F, and l is undefined in M	$\emptyset\Vert a \wedge (\neg b \vee c) \implies \neg b\Vert a \wedge (\neg b \vee c)$

Table 3.2: Classic DPLL Rules.

Name	Rule	Conditions
Learn	$M\Vert F \implies M\Vert F, C$	All atoms of C occur in F, $F \models C$
Forget	$M\Vert F, C \implies M\Vert F$	$F \models C$
Restart	$M\Vert F \implies \emptyset\Vert F$	

Table 3.3: CDCL DPLL Rules.

assignments away from the start the algorithm is. As the procedure assigns values to variables, a graph of the assignments and their implications can be built. When a conflict arises, a bipartition of the graph identifies the conflicting variables and the assignment that caused the implication. The way this information is stored and used for future assignments is implementation dependent but following the conflict, the algorithm will *backjump*, or backtrack non-chronologically, to the decision level before the assignment that caused the implication and will resume.

3.3 SMT

SMT solvers combine SAT solving with theory specific reasoning in order to tackle problems beyond simple Boolean formulas. While SAT solver problems are defined exclusively in propositional logic, SMT problems can be defined using first-order logic for any theories that the SMT solver supports. Commonly supported theories include the theory of uninterpreted functions, equality, bit vectors, arithmetic, and arrays. In practice these solvers look a lot like the automated reasoning machines that Hilbert and Leibniz imagined and can be applied to many domains such as program verification, scheduling, exploit generation, AI planning, and network analysis.

SMT solvers can be classified into two distinct types: eager solvers and lazy solvers. Eager solvers attempt to transform the theory expression into a Boolean formula that can be passed to the SMT solver's underlying SAT solver. Lazy solvers instead rely on their theory solvers and pass information between the theory solvers and the SAT solver. Although eager solvers, or bit blasters, are still commonly used for problems that manipulate bit-vectors, most SMT solvers, including leading SMT solvers Z3, CVC4, and Yices, now use

Name	Rule	Conditions
Theory Learn	$M\|\|F \implies M\|\|F, C$	All atoms of C occur in F, $F \models_T C$
Theory Forget	$M\|\|F, C \implies M\|\|F$	$F \models_T C$

Table 3.4: SMT Rules.

lazy solving /citeTinelli2012,smt-comp-results. Lazy solvers attempt to take advantage of the domain-specific strength of the theory solvers and rely only on the SAT solver for the skeleton Boolean formula avoiding the expensive transformation step [30, 34].

The adjustments that must be made to the state transition model of DPLL are as follows. The final state is either *unsat* or $M||F$ and $Solv_T(M)$ is satisfiable, where $Solv_T$ is a theory solver that can check theory satisfiability for F.

We also have to amend the Theory Learn and Forget rules to prevent being stuck in a state where $M||F, M \models F$, and $Solv_T(M)$ is not satisfiable by ensuring that the backtrack rule can be applied. The backtrack rule has a condition that $M \models \neg C$. In this unsatisfiable state $T \models \neg M$ so any clause for which $T \models C$ will enable backtracking. Finally, the pure literal rule needs to be removed. Although in Boolean logic literals are independent of each other in first-order logic this may not be the case.

3.3.1 Example Lazy Approach

Consider the problem below which uses the theory of equality of uninterpreted functions (EUF).

$f(a) = b \wedge (g(f(c)) \neq g(b) \vee g(a) = g(b)) \wedge a \neq c$

The SAT solver runs on the Boolean skeleton of the formula $1 \wedge (2 \vee 3) \wedge 4$ and suggests a model M, $1 = \mathit{true}, 2 = \mathit{false}, 3 = \mathit{true}, 4 = \mathit{true}$. The theory solver then finds the theory inconsistent, and adds a negation of the inconsistent model to the model preventing that state from being explored again - this new model $1 = \mathit{true}, 2 = \mathit{false}, 3 = \mathit{true}, 4 = \mathit{true}, (\neg 1 \vee 2 \vee \neg 3 \vee 4) = \mathit{true}$ is then solved by the SAT solver to produce a new model $1 = \mathit{true}, 2 = \mathit{true}, 3 = \mathit{false}, 4 = \mathit{true}$. This model is then sent to the theory solver which finds the theory consistent, producing a satisfiable result.

This basic lazy approach works well as it focuses on the domain specific strengths of the SAT solver and theory solver. It also allows the SMT solver to be built modularly as the SAT and theory solvers communicate via a simple interface enabling the theory or SAT solver to be replaced independently from each other. However, this basic lazy approach does not use any theory information to guide the search [30].

3.3.2 DPLL(T)

Rather than simply validating models suggested by the SAT solver, DPLL(T) uses the theory solver to find T-consequences to guide the search. It produces explanations of inconsistent states and is able to backtrack to known good states to continue the search [34, 35].

DPLL(T) adds a new rule T-Propagate which is described as $M||F \implies Ml||F$ under the conditions $M \models_T l$, l or $\neg l$ occurs in F, and l is undefined in M [30].

When a non-Boolean conflict occurs then the theory solver provides a conjunction, E, of literals that were in the model when it was found to be inconsistent. For each literal e in E,

Model	Formula	Rule Applied
	$f(a) = b \wedge (g(b) \neq g(f(a)) \vee c \neq a) \wedge f(c) = b$	UnitPropagate on $f(a) = b$
$f(a) = b$	$(g(b) \neq g(f(a)) \vee c \neq a)$	UnitPropagate on $f(c) = b$
$f(a) = b$, $f(c) = b$	$g(b) \neq g(f(a)) \vee c \neq a)$	T-Propagate on $\neg g(b) \neq g(f(a))$
$f(a) = b$, $f(c) = b$, $\neg g(b) \neq g(f(a))$	$c \neq a$	T-Propagate on $\neg c \neq a$
$f(a) = b$, $f(c) = b$, $\neg g(b) \neq g(f(a))$	*unsat*	

Table 3.5: DPLL(T) Results.

if e_1 was T-propagated then a reason for the inconsistency can be described as another set of literals R where $r_1 \wedge ... \wedge r_n \models_T e$.

Consider the unsatisfiable problem $f(a) = b \wedge (g(b) \neq g(f(a)) \vee c \neq a) \wedge f(c) = b$, the theory of EUF, and Table 3.5 which shows the steps performed in DPPL(T) to reach the unsatisfiable state. Initially the two unit literals are propagated resulting in a model of $f(a) = b$, $f(c) = b$, with this model the theory solver is able to T-propagate the negation of $\neg g(b) \neq g(f(a))$ as $\neg\neg g(b) \neq g(f(a))$ exists in the problem. Finally, $\neg c \neq a$ can be T-propagated which results in an unsatisfiable state.

3.3.3 Combining Theory Solvers

In many cases, problems cannot be simply modelled by a single theory. For instance, modelling arrays requires at least the theory of arrays and the theory of arithmetic. It may be possible to use the union of the theories using the Nelson-Oppen combination [34].

In order for two or more theories to be combined they must meet Nelson-Oppen restrictions. The classical restrictions are that the theories $T_1, ..., T_n$ must have disjoint symbols (excluding =), are convex, and are stably infinite. A theory is stably infinite if for every quantifier free satisfiable formula F described by T there is a $T - model$ that satisfies F which has an infinite model, examples of which include the theory of Arrays and EUF. A theory is convex if given a formula T-implies a disjunction of equalities, it also T-implies at least one of those equalities separately [36].

There are variations of the Nelson-Oppen combination for non-convex theories and non-stably infinite theories that have other restrictions such as the Shostak method [37].

Deterministic Nelson-Oppen Given two disjoint theories T_1 and T_2, where the theories are stably-infinite and convex, and a set of literals S, the deterministic Nelson-Oppen algorithm has the following steps.

1. Purify each literal in S so that it belongs to a single theory, S_1 or S_2 where $S = S_1 \bigcup S_2$

2. Agree constants for each literal in S and exchange equalities E, between S_1 and S_2

3. Check the satisfiability of each equality e in E with the corresponding solver. If any unsat, report unsat - else report sat.

For instance, consider the set of literals $f(a) - f(b) = c, c = a - 5, f(0) = 3, a = b$ and the theories of arithmetic and equality of uninterpreted functions.

Steps 1 and 2

$f(a) - f(b) = c \implies f(a) = t_1, f(b) = t_2, c = t1 - t2$

$c = a - 5 \implies t_5 = a - 5 \implies t_5 = t_3, t_3 = a - 5$

$f(0) = 3 \implies f(t_4) = t_5, t_4 = 0, t_5$

The solvers share constants: t_1, ... ,t_5, a, b, c.

Step 3

For each literal, its theory solver checks the satisfiability. In this example, each theory returns satisfiable.

Chapter 4: **JavaScript**

4.1 History

JavaScript was initially created by a team at Netscape in 1996 with the goal of bringing interactivity to the web via a scripting language. According to Brendan Eich (the creator of JavaScript), up until this point the web was “static, text-heavy, with at best images in tables or floating on the right or left” [1].

Brendan was initially recruited by Netscape with the aim of implementing Scheme, a dialect of Lisp that supports first-class functions [38], in the browser however by this time Sun and Netscape were already negotiating to bring Java to Netscape Navigator [39]. An internal debate occurred about the need for two languages but Eich and other influential developers, at both Sun and Netscape, believed that the two languages would serve different audiences. ‘Professional’ developers would veer towards Java for building logic heavy components and designers and amateurs would use the scripting language as a ‘glue’ for joining together components [1].

Despite targeting a different audience Mocha, the language that would later evolve into JavaScript, was required by management to “look like Java”, according to Eich, ruling out the existing languages Perl, Python, and Scheme. Eventually Eich settled on “Scheme-ish first-class functions and Self-ish (albeit singular) prototypes as the main ingredients” [39]. Mocha also inherited a number of confusing Java language features such as the distinction between primitives and objects (e.g. `string` vs. `String`) and the `Date` constructor which is a port of Java’s `java.util.Date`, complete with the Y2K bug [40]. Perl and Python are also credited to influencing Mocha’s string handling and regular expressions and AWK inspired the use of the `function` keyword [41]. The first version of the language had an incredibly short development period. Eich claims he spent “about ten days” developing the first JavaScript interpreter [1].

After JavaScript (abandoning its previous names Mocha and LiveScript) was released in Netscape Navigator 2 Microsoft began work on JScript, an equivalent language which shipped with Internet Explorer 3. Eich says that “At some point in late summer or early fall 1996, it became clear to me that JS was going to be standardized. Bill Gates was bitching about us changing JS all the time. [42]” This led to JavaScript being standardised by Ecma International, an industry group that produces information standards, as ECMAScript in 1997. Since 1997 ECMAScript, now in its sixth version, has also been adopted as an ISO/IEC standard [43].

A key moment in JavaScript’s history occurred when web developers became fascinated with Ajax, a set of techniques and technologies for making interactive websites, popularised by Google Suggest and Google Maps, spurring on an advancement in interactivity of modern websites [44]. To alleviate the pain of manipulating the Document Object Model (DOM), the hierarchy of objects that make up a HTML web page, and to deal with the problem that ‘writing JavaScript should be fun’ jQuery was released in 2006 [45] and commanded immediate popularity. In 2007, large tech companies including Digg, Google, Intel, Amazon, and the BBC all reported using jQuery [45] and in the 12 months between Sept 2007 and 2008 jquery.com received 13.5 million unique visitors [46]. As early as 2006 it is clear that although JavaScript is a powerful and popular language, in part due to its low barrier of entry and its unique place as the default language of the web, there are frustrations with the difficulty in reasoning about and writing JavaScript.

Type	Details
Undefined	The type of variables before they are assigned a value.
Null	The type of the assignable singleton value `null`.
Boolean	A standard representation of true and false.
String	An ordered sequence of zero or more 16-bit unsigned integers, these are intended to represent UTF-16 characters but are not required to be.
Number	Represents the IEE 754 double precision numbers in the range 2^{-253} to 2^{253} as well as the special cases NaN, ∞ , and $-\infty$. JavaScript's implementation of numbers includes both a positive and negative 0.
Object	Represents a collection of properties, each of which has a name and a value, as well as optionally a setter and/or getter function used to manipulate the value. To a first approximation, Objects can be thought of as a hash map.

Table 4.1: JavaScript Type System.

Google Chrome launched in 2008, with a new open source JavaScript engine named V8. When it launched V8 outperformed other browsers' JavaScript engines on benchmarks [47]. This was, in part, due to its hidden classes which reduce the cost of looking up properties on objects that share prototypes via inline caching, its use of a just-in time compiler (JIT) to produce assembly code rather than running an interpreter, as well as its efficient memory management [48, 49]. The launch of the V8 engine created a so called 'browser war' during which the performance of JavaScript vastly increased across all browsers [50].

In 2009, Node.js was launched - a JavaScript runtime based on V8 used to build asynchronous applications, popularising JavaScript as a language outside of the browser. Node.js was quickly adopted, with production applications at companies such as Uber and LinkedIn rolling out by 2011 [51, 52].

Due in part to popularity of Node, JavaScript is now one of the most popular languages for software development [2, 3] with npm, Node's package manager, hosting over 475,000 packages [53].

Despite its popularity, writing correct JavaScript code remains a relatively difficult problem largely due to its dynamic type system, its tendency to silently fail, and a number of quirks inherited from early versions of the language as a result of its short development cycle.

4.2 Type System

JavaScript has six types, as shown in Table 4.1, that variables can be [54], but the types of variables are not explicitly declared at compile time. Instead, types are dynamic and can change over the course of a program: `let foo = '1'; foo = foo * 2;`. Initially `foo` is of type String but the type is implicitly coerced to a Number by the `*` operator. Coercion between types occurs when the type of an operand is not the type the operator is expecting. For instance, as shown in the example above the `*` operator coerces all operands to numbers. This coercion is a source of error in programs as inputs of an invalid type can be coerced in unexpected ways.

Consider the contrived example in Figure 4.1 where the input passed to a function is a String instead of the expect Number. In this case the `+` operator coerces a Number to a String resulting in a concatenation rather than an addition.

```
function increment (x) {
  return x + 1;
}

let a = ’1’;
let b = increment(y); // b is the String ’11’ rather than the expected Number
    2
```

Figure 4.1: Example of Boolean Coercion.

Although the previous examples have used Strings and Numbers, type coercion extends to Boolean values as well. Non-boolean expressions are coerced with the following rules: `undefined`, `null`, `false`, the empty string, `NaN`, `+0`, and `-0` return `false`; while `true`, all other numbers, all other strings, and all objects return `true` [54]. These coercion rules, whilst a common source of error, can be taken advantage of to allow shortcuts like `if (x)` to check for both `undefined` and `null` values.

JavaScript also has some idiosyncrasies for testing equality which are underpinned by its type system. There are two operators for testing equality, `==` and `===`. The `==` operator, the ‘standard’ equality operator, uses coercion to test the equality of values of differing types. However the standard equality operator does not necessarily use the same coercion rules as other operators. For instance, whilst boolean coercion would typically coerce any non-zero and non-`NaN` number to `true`, `5 == true` is `false`. This inconsistency also extends to strings, typically all non-empty strings would coerce to `true`, i.e. the if case in `if (’3’){ ... }` would execute, but `’3’ == 1` is `false`. Additionally, if one compares an object to any other type standard equality converts the object to a primitive which can lead to some unexpected results such as `[’7’] == 7` being `true`. The strict operator, `===`, by comparison has much saner behaviour – any two values not of the same type are not equal.

4.3 Objects and Prototypal Inheritance

Typical object-oriented languages define classes which guarantee the exact sets of fields and methods an instance of the class will possess. JavaScript instead uses prototypal inheritance, in which objects have a *prototype* which defines a set of properties an object has, but objects are also free to declare their own set of properties and even overwrite their prototype’s properties.

Consider the example in Figure 4.2, in which we create a prototype, define a new object of that prototype, and then overwrite one of the values of the prototype.

A prototype may itself have a prototype, the sequence of prototypes which define the properties of a given object are referred to as a prototype chain [54, 55]. By default, all created objects inherit from `Object.prototype` and functions, as a special class of object, inherit from `Function.prototype` which inherits from `Object.prototype`. These prototypes are part of the EMCAScript standard and provide utilities to make working with objects of that prototype easier. For example, `Object.prototype` provides functions for iterating over all the values of an object or all the keys in the object.

Consider the prototype chain in Figure 4.4 in which we have a prototype chain length of three. Our `orange` object has a prototype of `Fruit`, which has a prototype of `Food`, which has a prototype of `Object`.

```
/* This is a constructor, color and weight will be values
 that belong the constructed object */
function Fruit (color, weight) {
  this.color = color;
  this.weight = weight;
}

// This is the set of values all fruit will inherit from the prototype
Fruit.prototype = {
  print: function () {
    console.log(‘Color: ${this.color} Weight: ${this.weight}‘);
  }
}

var orange = new Fruit(’orange’, 100); // orange.prototype === Fruit.prototype
orange.hasOwnProperty(’color’); // true, color belongs to orange
orange.hasOwnProperty(’print’); // false, print is inherited from orange.prototype
orange.print = () => console.log(’A Orange’); // Redefine the print function
orange.hasOwnProperty(’print’;) // true, print belongs to orange as well
```

Figure 4.2: Example of Prototypal Inheritance.

4.4 Arrays

Unlike other languages, JavaScript does not model arrays as continuously indexed tuples. Instead, arrays are a special form of object where the array elements are any value where the property can be coerced to a positive integer number less than 2^{32} - 1. These array elements are treated differently to regular properties by array prototypes and the length property.

The length of an array is a property of the array prototype which tracks the highest index in the array. Note, that the highest index of the array does not necessarily track the number of elements in the array; arrays do not enforce any kind of ordering on their properties, making it possible to create a non-contiguous array with ‘holes’ in it. For example `let arr = []; arr[0] = ’foo’; arr[2] = ’bar’;` produces an array of length 3. The length property is not read-only as one might expect. If one increases length, empty elements will be added to the end of the array, and decreasing the length will truncate the array to satisfy the new length.

Another by-product of arrays being a special form of object is that arrays are not homogeneously typed. Whilst most languages require arrays to only hold values of a single type, JavaScript objects and by extension arrays can contain multiple types of value, as shown by the following example: `let array = [’a’, 2.0, {}, new Fruit()]`.

As described above, arrays can have both properties and array elements. If the property fails the array element test described above, then the value is stored as a regular object value. This can lead to subtle bugs when working with numbers if bounds are not checked as values smaller than 0 and greater than 2^{32} - 1 will not be stored as an array element, as demonstrated in Figure 4.5.

```
function Food (calories, portions) {
  this.calories = calories;
  this.portionsPerDay = portions;
}

Food.prototype = {
  caloriesPerDay: function () {
    return this.calories * this.portionsPerDay;
  }
}

function Fruit (color, weight, calories, portions) {
  Food.call(this, calories, portions); // Call the food constructor
  this.color = color;
  this.weight = weight;
}

// The prototype of Fruit is Food, but we want to use the Fruit constructor
Fruit.prototype = Object.create(Food.prototype);
Fruit.prototype.constructor = Fruit;

// Let’s add our print function to the Fruit prototype
Fruit.prototype.print = function () {
  console.log(‘Color: ${this.color} Weight: ${this.weight}‘);
}

let orange = new Fruit(’orange’, 100, 47, 5);
orange instanceof Fruit; // true
orange instanceof Food; // true
orange instanceof Object; // true
```

Figure 4.3: Example of Prototypal Inheritance.

4.4.1 Prototype Functions

The array prototype has a large number of helper functions which developers frequently take advantage of, a number of which were added in the latest version of the EMCAScript standard [54]. Table 4.2 lists some of the more commonly used and interesting functions. As support for the latest ECMAScript standard varies between browsers it can be difficult to developers to judge when it is safe to use a new prototype function however using modern prototype functions typically makes reasoning about code easier. To resolve this tension, developers typically include *polyfills* – code which copies the behaviour of a prototype function and dynamically adds itself to the prototype if the function is `undefined`. Figure 4.6 contains a naive example implementation of a `Array.prototype.includes` polyfill.

4.5 Ensuring Correctness

Assuring the correctness of JavaScript code is an unsolved problem - in part due to the fact that JavaScript is a dynamically typed scripting language but also due to legacy design decisions. Efforts have been made to make the language itself safer to use, for example

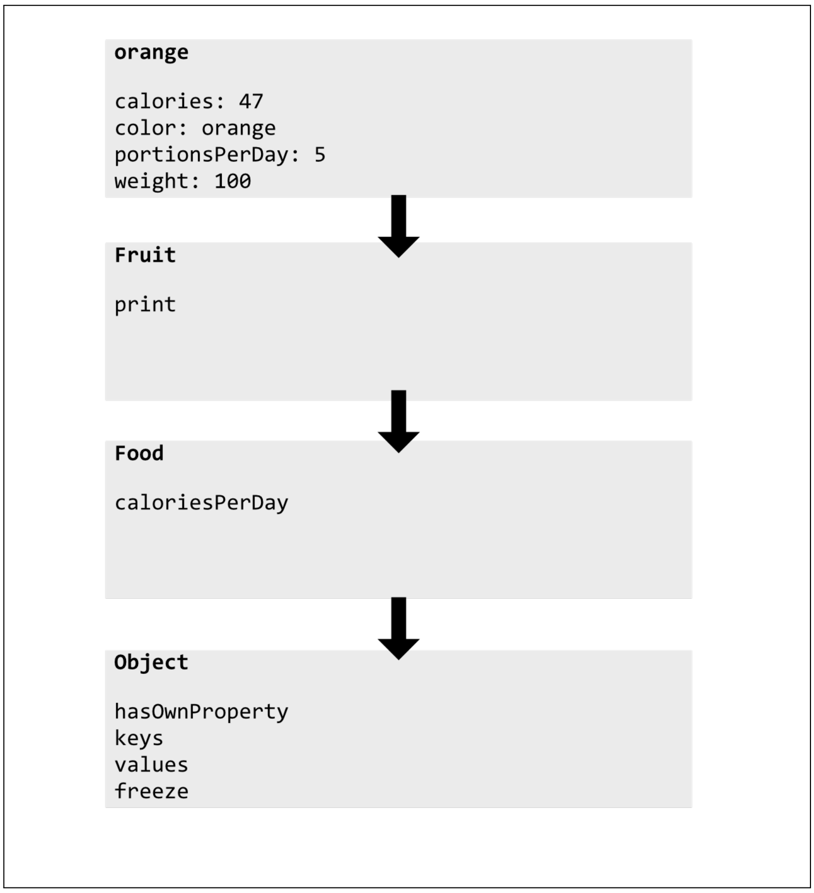

Figure 4.4: Prototypal Chain for Figure 4.3.

```
let arr = [];
arr[0] = 'foo'; // array is
arr.length; // returns 1
arr[4294967296] = 'bar'; // 4294967296 === 2^32-1, property not an element
arr[-1] = 'baz'; // -1 < 0, property not an array element
arr.length; // still returns 1
```

Figure 4.5: Example of Array Object Behaviour.

ECMAScript 5 introduced a strict mode (later made the default mode in ECMAScript 6) which restricts the use of certain language features and also defines additional circumstances in which exceptions should be thrown. However there are still many subtle errors that can occur even within the safer subset of JavaScript.

Tools like Eslint and Flow are commonly used by developers to statically analyse code and identify bugs but are limited in their scope. Eslint uses a narrow set of rules to identify possibly dangerous behaviour. Flow is constrained to reasoning about types and in some cases requires additional code annotation to identify possible errors. These limited sets of warnings do not fully reflect the dynamic nature of JavaScript and the subtle interactions that can occur due to prototypal inheritance, dynamic property access, and dynamic dispatch. There have been a number of attempts to analyse JavaScript statically however most approaches fall short of useful due to their inability to reason about functions like `call()` and `apply()` [56], with the best results requiring an element of dynamic analysis as well [57, 58].

Consequently there has been a shift in interest towards analysing safer (and less dynamic) targets which compile to JavaScript, the most popular of which is Typescript. Typescript

```
if (!Array.prototype.includes) {
  Object.defineProperty(Array.prototype, 'includes', {
    value: function(needle, index) {
      if (this == null) {
        return false;
      }

      index = index || 0;
      return this.indexOf(needle) > index;
    }
  });
}
```

Figure 4.6: Example of a Naive Includes Polyfill.

is a superset of ECMAScript 6 which includes optional types to enable errors to be caught statically before compilation. Although TypeScript makes it easier to reason about potential errors caused by coercion, TypeScript itself provides no guarantees of static soundness - it is still possible for a TypeScript program to compile to JavaScript successfully and encounter a run-time error as a result of type coercion when executed [59]. Extensions to TypeScript have been suggested to provide soundness [60, 61] although these approaches require a modified interpreter or runtime enforcement. Regardless, whilst TypeScript and its derivatives provide a way of reducing or eliminating type based errors they do not help identify or eliminate errors that occur due to other dynamic features of JavaScript such as prototypal inheritance, dynamic property access, or dynamic dispatch.

There have been a number of attempts to produce formal semantics for JavaScript, to allow for better reasoning about JavaScript code, however no attempts can be considered completely successful. One approach taken is to translate a subset of the JavaScript language (targeting ECMAScript 5 in strict mode) to an intermediate language, JSL, on which analysis can be performed [62]. Although the results of this type of analysis is promising, the process and toolchain requires an expert level understanding of the semantics of JavaScript and also requires developers to annotate their code pre and post-conditions. While, this approach may be suited for smaller programs (or critical subsets of programs) its approach does not scale well to large applications. Other approaches such a KJS [63] and λJS[64] support even less of the language than JaVerT and crucially none of the three approaches target, or can be easily extended to, the ECMAScript 6 specification.

Symbolic execution is a good match for JavaScript. By concretely executing the program, reasoning about the dynamic nature of JavaScript is deferred to the interpreter. The use of an interpreter for testing also provides flexibility. If JavaScript needs to be tested in another environment, switching interpreter is a relatively easy change to make. Additionally, concrete execution comes without the penalty of having to model the entire language specification as any non-modelled function can simply revert to concrete execution.

Function Name	Signature	Description
`indexOf`	`indexOf(element)`	Returns the first index of the array that contains `element` or -1 if `element` is not in the array
`lastIndexOf`	`lastIndexOf(element)`	Returns the last index of the array that contains `element` or -1 if `element` is not in the array
`slice`	`slice(begin, end)`	Returns a copy of the array with the elements from the index `begin` up to the `end` index. If `end` is not provided it will slice `from begin` to the last element of the array
`push`	`push(element)`	Increases the length of the array by one and adds `element` to the end of the array
`pop`	`pop()`	Removes the last `element` in the array and returns it
`unShift`	`unShift(elementA, elementB, ...)`	Adds one or more elements to the beginning of an array and returns the new length
`includes`	`includes(element, startIndex)`	Returns `true` if the array contains `element` either starting from `startIndex` or the start of the array if no `startIndex` is provided
`reverse`	`reverse()`	Reverses the array in place and returns a reference to the array
`forEach`	`forEach(func)`	Calls `func(value, index, array)` on each element in the array.
`filter`	`filter(func)`	Returns a new array that contains elements that return `true` when `func(currentElement)` is called on them.
`map`	`map(func)`	Returns a new array with the results of calling `func( value, index, array)` on each element in the array.
`reduce`	`reduce(func,initialValue)`	Calls `func(accumulator, value, index, array)` on every element in the array and returns the value of accumulator. If `initialValue` is provided then the first time `func` is called then `accumulator` is set to the value of `initialValue`
`some`	`some(func)`	Calls `func(element, index, array)` for every element in the array. Returns `true` if `func` returns `true` for at least one element in the array
`every`	`every(func)`	Calls `func(element, index, array)` for every element in the array. Returns `true` if `func` returns true for all elements in the array

Table 4.2: Array Prototype Functions.

Chapter 5: **Implementation**

ExpoSE [4] is a recently developed tool for dynamic symbolic execution of JavaScript. It takes a program as input and attempts explore all feasible paths in the program up to a maximum path count or until it exceeds a maximum time limit. Programs are instrumented using Jalangi2, a framework that provides callbacks for JavaScript statements which are used to build the symbolic representation of the program. New inputs are generated by solving path constraints using the Z3 SMT solver [65].

Below I outline the implementation of my extensions to ExpoSE to support arrays. Specifically I describe the challenges in encoding JavaScript arrays in Z3, the modifications necessary to symbolically reason about arrays and their length, and a description of the function models I have written for Array prototype functions.

5.1 JavaScript Array Encoding

Z3 supports two array-like theories: the theory of arrays and the theory of sequences which provided a choice for which theory to encode JavaScript arrays in. Z3 offers a rich number of operations on sequences including obtaining the length, concatenating sequences, and checking whether an element is in a sequence. Conversely, the theory of array support is limited to three basic operations: select, store, and map. Initially sequences seemed like a more natural fit as many array prototype functions map naturally to Z3 sequence operations. For instance, `Z3_mk_seq_contains` is very similar to `Array.prototype.includes` and `Z3_mk_seq_index` is similar to to `Array.prototype.indexOf`. However after experimentation it became clear that concretising sequences was impossible as extracting single elements from a sequence produced a unit sequence rather than the value of the element. As a result of being unable to concretise sequences, JavaScript arrays are encoded using Z3s array theory.

Z3 arrays are one dimensional unbounded maps where the index and values both have a *sort*, Z3's terminology for type. Read and write operations on the array are mapped to select and store axioms respectively where the relationship between select and store is defined by $select(store(a, i, v), i) = v$ where a is an array, i is an array index, and v is a value. Due to their unbounded and strongly typed nature, Z3 arrays do not directly correspond to JavaScript arrays which, as described in Section 4.4, can contain multiple types, are bounded between the indices 0 and 2^{32}-1, and have a mutable length. As a result certain tradeoffs have to be made in the way arrays are modelled and encoded. The first limitation is that arrays are only able to be symbolically modelled if they are initialised with values of a single type and only if the type of the values have an analogous Z3 sort which limits the types that are supported to booleans, numbers, and strings. The second limitation is that holes that are created by mutating the length of an array or setting values non-sequentially are not modelled. Finally, in order to encode the finite length of JavaScript arrays, for each Z3 array a separate Z3 integer variable is required represents the length of the array.

5.1.1 Representing Array Length

When the array is created an assertion $Length > 0$ is made. Whenever the concrete JavaScript array is accessed, if the index, i, is within the bounds of the concrete array then an assertion $i > 0 \wedge i < Length$ is made, implying that the array's length is greater than the symbolic index. If the index is not within the bounds of the concrete array then the

```
1 function array_manipulation() {
2   var a = ['a']; // Constraint: Length0 >= 0
3   a[0]; // Constraint: Length0 > 0
4   a.push('b'); // Constraint: Length1 = Length0 + 1
5   a[5]; // Constraint: Length1 = Length0 + 1 ^ Length0 < 5
6   a[20] = 'c'; // Constraint: Length2 > 20
7   a.pop(); // Constraint: Length3 = Length2 - 1
8 }
```

Figure 5.1: Example of Array Constraints Enforcement.

assertion $\neg(i \geq 0 \wedge i < Length)$ is made, which implies that the symbolic length is less than the accessed index. When the array is pushed to and popped from, a new length variable is created and the length is asserted to be the previous length plus or minus one. When the array is set to, a fresh length variable is created and an assertion $Length < i$ is made. When the array length is directly set to, `{array.length = x;` a fresh length variable is created and an assertion $Length = x$ is made, where x is the value that was set.

Consider the example program in Figure 5.1 which demonstrates the constraints on the symbolic length of an array as it is accessed and updated. Initially, (line 2) the length of the array is constrained only to be greater or equal to 0, effectively unbounded. On line 3, the array is accessed at index 0 and the constraint then changes to have a length greater than the accessed value as the concrete array (`['a']`) has an item at that index. On line 4, the array is pushed to and a new Z3 variable is created to model the length of the array but the constraint still depends on the previous variable - the new constraint is the previous variable's constraints plus one. On line 5 the array is accessed out of bounds and a new assertion is made that the length is less than the out of bounds index. When the array is set on line 6 another fresh Z3 variable is used to model the length of the array but this does not depend on the previous variable, instead it asserts that the length is greater than the accessed index (20). On line 7 the array is popped from, and similarly to line 4 when the array was pushed, a new length variable is created than depends on the previous variable's constraints minus one.

5.2 Array Support Architecture

ExpoSE is architected as a number of distinct modules: Z3JS, a JavaScript interface to Z3s C API, the Distributor, a highly parallelised dispatcher for symbolic execution paths and an aggregator of information, the Analyser, which executes the symbolic analysis for a given path, and the Tester, which automatically runs ExpoSEs unit tests. The bulk of my work, was done in the Analyser and Z3JS. Below I briefly outline the modifications that were made to each.

5.2.1 Z3JS

Z3JS provides an interface between the Z3s C API and JavaScript. Its main components are a `Solver` class, a `Query` class, a `Model` class, an `Expr` class, and a `Context` class. The `Context` class wraps the Z3 context and acts as a facade for the Z3 API calls. The `Expr` class wraps constructed Z3 expressions and has helper functions for printing and creating a concrete value from the symbolic expression. The `Model` class wraps Z3 model objects and the query class is used to request solutions for models from Z3.

In order to be able to interface with Z3s array functions I had to add a number of functions to the context class including `mkArray`, `mkSelect`, `mkStore` as well as functions used to perform assertions using quantifiers, `mkForAll` and `mkExists`. The quantifier based functions expect arrays to be passed as parameters so I also had to change some helper functions `_buildChecks()` and `Z3Utils.astArray()` to work with arrays of parameters.

I modified `Expr` to a hold of number of array specific properties that need to be required throughout ExpoSE such as the expression for array length, the type of the array, the start index of the array (described in detail in Section 5.2.3), the name of the array, and a counter for the number of length expressions an array has. All of these properties have getter functions and either an increment function or a setter. I have also written helper functions to hide the complexity of storing to `setAtIndex()` and selecting from (`selectFromIndex()`) an array expression.

A key function in `Expr` is `asConstant()` which turns the expression into a concrete value. I rewrote the function into a small switch statement which evaluates the sort of the expression. To be able to support turning array expressions into constants I had to be able to get a constant value for the expression which represents the length of the array. In order to do that I had to change the signature of the function to accept a model, using which I can call `model.eval()` to get an interpretation for the expression. Using the concrete value for the length I can then iterate for the length of the array calling `selectFromIndex()` to produce a concrete JavaScript array of values.

5.2.2 Analyser

SymbolicState.js has a number of functions that are called when a symbolic value is used during execution. `createSymbolicValue()` has been modified to check if the concrete type is either homogeneously typed or an empty array, if so Z3JSs `context.mkArray()` is called to create the array, with the correct homogeneous type, and assertions are made about the arrays length, as described above, to keep the array bounded.

`symbolicField()` has been modified to check if the array is being accessed with a potentially valid array index - i.e. a number between 0 and 2^{32}-1. If so, assertions are made about the length, as described above, and if the index is less than the array length Z3JSs `Expr.selectFromIndex()` is called. The assertions about the array length are pushed using `symbolicConditional()` which pushes either an expression, or the negation of that expression, depending on the result of a concrete condition - consequently whenever an in-bounds array access occurs, `let arr = [0, 1, 2]; arr[1];`, a new path will be created to explore an out of bounds access and vice versa. `symbolicField()` has also been modified to return the expression for the array's length when the `length` property is accessed.

`putField()` has been modified, in SymbolicExecution.js, to call `symbolicSetField()`, a new function in SymbolicState.js, if the concrete value `putField()` is called on is an array. `symbolicSetField()` has two cases: if the field is a valid array index then the value is stored in the array and a new length variable is create, alternatively if the `.length` field is assigned to then the value is set as the new length.

5.2.3 Function Models

When symbolic values are used in native

```
function includes(needle) {
  var array = this;
  for (var i = 0; i < array.length; i ++) {
    if (array[i] === needle) {
      return true;
    }
  }
  return false;
}
```

Figure 5.2: Example of a Naive Implementation of a Function Model for Includes.

ExpoSE has a framework for modelling prototype functions in the `FunctionModel` module. Prototypes in the `FunctionModel` module are exposed as a symbolic hook that concretely executes the prototype function and then if a condition is met also symbolically executes a model function if it exists.

Below I provide a detailed look at two of the function models that have been implemented `slice` and `pop`.

slice

`Array.prototype.slice` returns a shallow copy of a subsection of array. It takes two optional parameters `begin` which marks the start index for the shallow copy to begin from and `end` which marks the first index in the array that should not be included in the shallow copy. Both parameters also accept negative indices which are interpreted as reverse indexing. If no `begin` parameter is passed then `slice` returns a full copy of the array.

The function model for `slice` is shown in Figure 5.3. The first function passed to the `symbolicHook` checks that the function the prototype is called on is a symbolic array as the `slice` function can also be used on strings. The second function passed to the `symbolicHook` models the function. Lines 4-18 check which arguments are passed to the function, set the default parameters if necessary, and perform validation. Validation is performed by asserting that if the first argument is negative that `begin` should be 0 and `end` should be set to the value of the first argument, as this is a negative indexing operation. Lines 21-22 create a shallow copy of the array by performing a store with no effect and lines 24-26 create a new variable to represent the new array's length. Line 26 increases the *start index* of the copied array - the start index is the pointer to the first index of the unbounded Z3 which should be considered part of the array that is being symbolically modelled. This methodology allows for an arbitrary number of slices to be made by continually incrementing the pointer. Both `Expr.selectFromIndex()` and `Expr.setAtIndex()` use the start index offset hiding the complexity of accounting for the start index pointer in most cases but in the `FunctionModels` module, where the helper functions, the start index needs to be taken into account. Lines 31-42 describe assertions on the length of the new array. The length must be greater or equal to 0 and if `end` is positive, the length of the copied array is $end - start$, else the new array length is $oldArrayLength - (end + begin)$.

This model allows for all but one of the acceptable signatures of `slice` including `slice()`, `slice(-1)`, and `slice(1, -2)` but excluding `slice(4, 1)` – which would return an empty array.

pop

`Array.prototype.pop` returns the element at the last index of the array and reduces its length by one. The function model for `pop` is shown in Figure 5.4. Line 2, included in every function model, checks that the `base` argument is symbolic and an array - if so the second function starting on line 3 is called. Lines 4-8 create a new Z3 variable to store the new length, and create a temporary variable to store the old length. Line 10 fetches the value to be popped from the array and lines 11-13 set the new length to one less than the previous length and assign it to the array expression. The popped value is then returned.

5.2.4 Tests

All of the array functionality implemented, including function models and the regular concrete behaviour of arrays, is backed by a number of tests. Tests are written in a way to prompt to ExpoSE to attempt to generate inputs which would explore infeasible paths, as demonstrated in Figure 5.5. The expected number of errors are recorded in an array specific test runner and if the number of errors thrown does not match then the test runner rethrows the error indicating a change in behaviour. The array-specific test suite is run as a git pre-push hook to act as a smoke test and prevent regressions being committed to the repository.

```
models[Array.prototype.slice] = symbolicHook(
            (c, _f, base, args, _r) => c.state.isSymbolic(base) && c.state.getConcrete(base) instanceof Array,
            (c, _f, base, args, result) => {
                const ctx = c.state.ctx;
                const array = c.state.asSymbolic(base);
                let begin = args[0] ? c.state.asSymbolic(args[0]) : ctx.mkIntVal(0);
                let end = args[1] ? c.state.asSymbolic(args[1]) : array.getLength();

                // handle slice(-1)
                // if begin is less than 0, begin equals 0 and end equals arg[0], else begin must be ge than 0
                c.state.pushCondition(ctx.mkIte(
                    ctx.mkLt(begin, ctx.mkIntVal(0)),
                    ctx.mkAnd(ctx.mkEq(begin, ctx.mkIntVal(0)), ctx.mkEq(end, c.state.asSymbolic(args[0]))),
                    ctx.mkGe(begin, ctx.mkIntVal(0))
                ));

                // end cannot be greater than array length
                c.state.pushCondition(ctx.mkLe(end, array.getLength()), true);

                // This clones the array in Z3 by doing a store that stores the same value at the index
                const noOpIndex = ctx.mkIntVal(0);
                const copiedArray = array.setAtIndex(noOpIndex, array.selectFromIndex(noOpIndex));

                const newLength = ctx.mkIntVar(`${array.getName()}_Length_${array.incrementLengthCounter()}`);
                copiedArray.setLength(newLength);
                copiedArray.increaseStartIndex(begin);

                /* The new length should be greater than 0 AND
                   IF end is positive, less than or equal to end's length -> slice(0, 4)
                   ELSE, less than or equal to the old length-end -> slice(0, -1) */
                c.state.pushCondition(
                    ctx.mkAnd(
                        ctx.mkGe(newLength, ctx.mkIntVal(0)),
                        ctx.mkIte(
                            ctx.mkGe(end, ctx.mkIntVal(0)),
                            ctx.mkLe(newLength, ctx.mkSub(array.getLength(), end)),
                            // negative end indexing
                            ctx.mkLe(newLength, ctx.mkSub(array.getLength(), ctx.mkAdd(end, begin)))
                        )
                    ),
                    true
                );
                return new ConcolicValue(result, copiedArray);
            }
        );
```

```
models[Array.prototype.pop] = symbolicHook(
            (c, _f, base, args, _r) => c.state.isSymbolic(base) && c.state.
                getConcrete(base) instanceof Array,
            (c, _f, base, args, result) => {
                const ctx = c.state.ctx;
                const array = c.state.asSymbolic(base);

                const oldLength = array.getLength();
                const newLength = ctx.mkIntVar(‘${array.getName()}_Length_${
                    array.incrementLengthCounter()}‘);

                const value = new ConcolicValue(result, array.selectFromIndex
                    (oldLength));
                c.state.pushCondition(ctx.mkEq(newLength, ctx.mkSub(oldLength
                    , ctx.mkIntVal(1))), true);

                array.setLength(newLength);
                return value;
            }
        );
```

Figure 5.4: Pop Function Model.

```
var q = symbolic UnderTest initial [0, 1, 1, 4, 4, 1];

// Clone the length
var length = q.length;
if (q.length === 3) {
  q.length = 42;
  if (q.length === 42) {
    console.log(’Success’);
  } else {
    throw ’array_set_length: This should be unreachable’;
  }
}
```

Figure 5.5: Example Unit Test.

Chapter 6: **Evaluation**

To evaluate the extensions to ExpoSE I have tested two popular JavaScript libraries, `Lodash/array` and `Minimist`. Both libraries have roughly 14 million downloads a week from npm and make extensive use of array operations making them ideal targets to evaluate. I also present a methodology for verification of array polyfills using ExpoSE using which I discover bugs in a popular polyfill library.

6.1 Methodology

Using a version of ExpoSE with support for numbers, strings, Booleans, `undefined`, `null` and the extensions for array support described in Section 5 I symbolically executed `Minimist` and `Lodash/array` with and without array support enabled. `Minimist` was tested with a simple harness as the module is exported as a single function and `Lodash/array` was tested with ExpoSE's harness generator which explores all the public functions in a library with symbolic arguments of all supported types. Tests were run on the same machine with 264GB of RAM and an Intel Xeon 6142 @ 2.6GHz CPU with 64 cores. Tests were left to run until they exhausted all paths, hit a max path count of 1,000,000 or ran for 15 minutes. Impact is measured in the amount of coverage gained with array support enabled, the number of additional paths generated, as well as the increase in time taken.

6.2 Results

The results in Table 6.1 show a clear difference in behaviour of the two libraries that are tested, although both have a net gain in coverage. `Minimist` has a drastic jump in coverage with array support enabled which comes at the cost of total path count. The number of paths appears to be reduced as the execution was constrained by time, rather than a reduction in a number of feasible paths. The combination of a low path count and the high max and 90^{th} percentile test case duration indicates a small number of large queries which suggests that ExpoSE had large path constraints, potentially as a result of deeper exploration of the program. `Lodash/array` exhibited different behaviour, whilst it only received a small bump in coverage it more than doubled the number of paths explored and spent on average 30% longer on each path.

Examining why the jump in coverage was so small revealed that the majority of functions accept strings. The remaining uncovered areas of `Lodash/array` mostly relate to internal functions which are unreachable from the `Lodash/array` module and polyfill code. Both libraries have areas that concretise due to lack of Object support.

Overall the increased cost in query time appears to be a reasonable trade-off for the net gain in terms of both coverage and additional paths to explore although there is reason to be concerned about the performance of arrays on particular deep paths with many assertions as demonstrated by the large increase (100s) in max test case duration of `Minimist`.

	`Minimist` with Arrays Disabled	`Minimist`	`Lodash/array` with Arrays Disabled	`Lodash/array`
Lines Covered	34%	73%	82.6%	83%
Total Test Duration	902s	901s	376s	801s
Median Test Case Duration	6.49s	5.42s	7.17s	9.63s
90th Percentile Test Case Duration	9.7s	21.93s	10.75s	11.55s
Max Test Case Duration	215.93s	315.33s	14.8s	17.41s
Total Paths Explored	7243	2015	2450	5101

Table 6.1: Results.

6.3 Polyfill Verification

Array polyfills, as discussed in 4.4.1, attempt to mimic the behaviour of standard prototype functions for backwards compatibility purposes. By giving symbolic input to both a polyfill and the standard implementation function and comparing the result deviations in behaviour can be identified. Below I present an example harness and describe a bug found in a popular polyfill library.

The harness, shown in 6.1, calls the function `testPolyfill()` with the polyfill function under test, the matching standard function, a symbolic `thisArg` argument, and an array of symbolic values. The `thisArg` argument is the argument that represents the value that the prototype is being invoked from, the `arr` in `arr.includes(myValue, 0);`. The array of symbolic values are the values that should be passed to the function. `testPolyfill()` then uses `apply()` to call the two functions, using `try` to prevent an exception from causing the test case to fail, and stores the results. If the results differ then an exception is thrown which contains a string representation of the two values and the key word alert to make the output easily to identify using `grep`.

`js-polyfills` is a popular polyfill library with 970 stars on Github and 7,000 weekly downloads. It contains polyfills for many array functions including `Array.includes`. Using the harness and techniques described above, ExpoSE identified that calling the `includes` polyfill with an `undefined` base type would cause the function to return `false` whereas the standard prototype implementation throws an exception. If a program used a pattern like `try {array.includes(a) } catch (e){...}` to do error handling in the case of being passed an undefined array then as a result of this bug the program would function differently in a polyfilled environment.

```
"use strict";

function testPolyfill(testFunction, prototypeFunction, base, args) {
    console.log('Base ' + base + ', args ' + args);

    var result1;
    var result2;

    try {
      result1 = testFunction.apply(base, args);
    } catch(e) {
      // Empty catch
    }

    try {
      result2 = prototypeFunction.apply(base, args);
    } catch(e) {
      // Empty catch
    }

    result1 = JSON.stringify(result1);
    result2 = JSON.stringify(result2);

    if (result1 !== result2) {
      throw 'Alert -> Error: polyfill: \'' + result1 + '\' prototype: \'' +
          result2 + '\'';
    }
  };

var from = require('array-from');
testPolyfill(from, Array.from, symbolic thisArg, [symbolic Needle]);
```

Figure 6.1: Polyfill Test Harness.

Chapter 7: **Conclusion**

In this report I have presented an encoding of JavaScript arrays using the Z3 SMT solver. I also discussed the implementation of extensions to ExpoSE to support arrays including example models for popular array prototype functions. I also demonstrated that the extensions to ExpoSE are effective at increasing coverage on programs that use arrays with a reasonable increase in time spent solving and demonstrate a case where as a result of the implementation ExpoSE is able to find a bug in a popular polyfill library.

There are many possible extensions to the work presented in this report. The most natural extension would be to explore supporting non-homogeneous arrays, which could be achieved through encoding an array as seven Z3 boolean arrays, one for each JavaScript type plus an additional array to encode which array an element exists in. This approach leans heavily on Z3 and would likely come with a large performance cost. Another possible extension would be to extend homogeneous arrays to support holes, a naive approach would be to store a 'magic value' at indices that should contain a hole however this approach would only work for String and Number arrays. A more advanced approach could add an uninterpreted function to array expressions which would indicate the indices in the array that should contain holes. An additional avenue of future work could be to extend pure symbolic symbols to use an array of each supported type and an empty array as opposed to just an array of strings.

If I were to work on a similar project in the future I would attempt to use real-world-esque programs sooner in my analysis. Whilst throughout the project's lifespan I've had a number of small (5-15 line) test cases there were a number of bugs that did not become apparent until I tested my array implementation on a real world target. To combat this, I would write a number of non-trivial programs in the range of 50-200 lines of code that would have well defined expected behaviour. I would also write code more defensively and with more verbose logging. I encountered two major bugs which I invested several days into which could have been prevented if I had. The first bug was that `Array.prototype.slice` is also used reused for strings however I was not checking in my prototype functions that the `this` argument was actually an array, had I written my code more defensively then this would not have been an issue. Another bug I identified was that my put field method was silently failing to be called as it was being called with concrete arguments – a more liberal use of logging would have made this much easier to track down. JavaScript is a language that has a lot of edge cases and writing code assuming it may fail seems to be the only sensible way to protect against unexpected behaviour.

Overall I believe the project was a success and I am proud of the contributions I have presented.

Bibliography

[1] "The A-Z of programming languages: JavaScript.." https://web.archive.org/web/20180210015852/https://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/, July 2008. Accessed: 2010-09-30.

[2] "Stack Overflow developer survey 2017." https://insights.stackoverflow.com/survey/2017#most-popular-technologies, 2017. Accessed: 2010-09-30.

[3] "TIOBE index for November 2017." https://www.tiobe.com/tiobe-index/, Nov. 2017. Accessed: 2010-09-30.

[4] B. Loring, D. Mitchell, and J. Kinder, "ExpoSE: practical symbolic execution of standalone JavaScript," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017* [4], pp. 196–199.

[5] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[6] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT - a formal system for testing and debugging programs by symbolic execution," in *Proceedings of the International Conference on Reliable Software*, (New York, NY, USA), pp. 234–245, ACM, 1975.

[7] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing.," in *NDSS*, vol. 8, pp. 151–166, 2008.

[8] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), pp. 213–223, ACM, 2005.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.

[10] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[11] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pp. 206–215, 2008.

[12] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[13] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 571–572, ACM, 2007.

[14] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pp. 263–272, 2005.

[15] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209–224, 2008.

[16] A. Almossawi, K. Lim, and T. Sinha, "Analysis tool evaluation: Coverity prevent," *Pittsburgh, PA: Carnegie Mellon University*, pp. 7–11, 2006.

[17] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, pp. 66–75, Feb. 2010.

[18] R. P. Jetley, P. L. Jones, and P. Anderson, "Static analysis of medical device software using CodeSonar," in *Proceedings of the 2008 workshop on Static analysis*, pp. 22–29, ACM, 2008.

[19] C. Barrett, "SMT: Where do we go from here?." Presentation `http://smt2014.it.uu.se/slides/barrett2.pdf`, July 2014.

[20] A.-J. Kaijanaho, "Theory of automated reasoning: An introduction," 2004.

[21] A. Pitts, "Computation theory." Presentation `https://www.cl.cam.ac.uk/teaching/0910/CompTheory/lectures/lecture-1.pdf`.

[22] C. Barrett, "The satisfiability revolution and the rise of SMT." Presentation `https://excape.cis.upenn.edu/documents/ClarkBarrettSlides.pdf`, Mar. 2014.

[23] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, pp. 201–215, July 1960.

[24] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.

[25] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM (JACM)*, vol. 12, no. 1, pp. 23–41, 1965.

[26] A. Newell and H. Simon, "The logic theory machine–a complex information processing system," *IRE Transactions on information theory*, vol. 2, no. 3, pp. 61–79, 1956.

[27] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th annual Design Automation Conference*, pp. 530–535, ACM, 2001.

[28] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pp. 279–285, IEEE Press, 2001.

[29] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability solvers," *Foundations of Artificial Intelligence*, vol. 3, pp. 89–134, 2008.

[30] C. Barrett, "From SAT to SMT: The DPLL(T) architecture." Presentation `https://web.stanford.edu/class/cs357/lectures/lec9.pdf`.

[31] "The Davis-Putnam-Logemann-Loveland procedure." `https://www.cs.utexas.edu/users/vl/teaching/lbai/dpll.pdf`, 2011. Accessed: 2018-03-22.

[32] A. Farinell, "DPLL method." Presentation `http://profs.sci.univr.it/~farinelli/courses/ar/slides/DPLL.pdf`, 2010.

[33] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann–Loveland procedure to DPLL (T)," *Journal of the ACM (JACM)*, vol. 53, no. 6, pp. 937–977, 2006.

[34] A. Oliveras, "SMT theory and DPLL(T)." Presentation `http://sei.pku.edu.cn/~xiongyf04/SA/2014/5_SMT.pdf`, July 2013.

[35] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL (T): Fast decision procedures," in *International Conference on Computer Aided Verification*, pp. 175–188, Springer, 2004.

[36] E. Torlak, "Combining theories." Presentation `https://courses.cs.washington.edu/courses/cse507/14au/slides/L7.pdf`.

[37] Z. Manna and C. G. Zarba, "Combining decision procedures," in *Formal Methods at the Crossroads. From Panacea to Foundational Support*, pp. 381–422, Springer, 2003.

[38] R. K. Dybvig, *The Scheme Programming Language: ANSI Scheme.* Prentice Hall PTR, 1996.

[39] B. Eich, "Popularity." `https://brendaneich.com/2008/04/popularity/`, Apr. 2008. Accessed 2017-01-01.

[40] "Every day I learn something new... and stupid." `https://www.jwz.org/blog/2010/10/every-day-i-learn-something-new-and-stupid/#comment-1048`, Apr. 2011. Accessed 2017-12-01.

[41] B. Eich, "A brief history of JavaScript." `https://brendaneich.com/2010/07/a-brief-history-of-javascript/`, July 2010. Accessed 2017-12-01.

[42] B. Eich, "New JavaScript engine module owner." `https://brendaneich.com/2011/06/new-javascript-engine-module-owner/`, June 2011. Accessed 2017-12-01.

[43] I. J. S. 22, "Information technology – programming languages, their environments and system software interfaces – ECMAScript language specification," standard, International Organization for Standardization, Geneva, CH, June 2011.

[44] J. J. Garrett, "Ajax: A new approach to web applications." `http://adaptivepath.org/ideas/ajax-new-approach-web-applications/`, Feb. 2005. Accessed 2017-12-01.

[45] Jeresig, "History of jQuery." `https://www.slideshare.net/jeresig/history-of-jquery`, Oct. 2007. Accessed 2017-01-01.

[46] Jeresig, "State of jQuery '08." `https://www.slideshare.net/jeresig/state-of-jquery-08-presentation/2-Growth_Huge_growth_in_2008`, Oct. 2008. 2017-01-01.

[47] R. Goodwins, "Google Chrome - first benchmarks. summary: wow.." `https://web.archive.org/web/20080903125550/http://community.zdnet.co.uk/blog/0%2C1000000567%2C10009139o-2000331777b%2C00.htm`, Sept. 2008. Accessed 2017-01-01.

[48] L. Bak, "V8: an open source JavaScript engine." Video `https://www.youtube.com/watch?v=hWhMKalEicY`, Sept. 2008. Accessed 2017-12-01.

[49] M. S. Ager, "Chrome: V8 engine." Presentation `https://www.youtube.com/watch?v=JxUvULKf6A4`, Nov. 2008.

[50] D. Mandelin, "Know your engines: How to make your JavaScript fast." `https://cdn.oreillystatic.com/en/assets/1/event/60/Know%20Your%20Engines_%20How%20to%20Make%20Your%20JavaScript%20Fast%20Presentation%201.pdf`, June 2011. Accessed: 2017-12-01.

[51] J. O'Dell, "Exclusive: How LinkedIn used Node.js and HTML5 to build a better, faster app." `https://venturebeat.com/2011/08/16/linkedin-node/`, Aug. 20111. Accessed: 2017-12-01.

[52] C. Chambers, "Node.js meetup: Distributive web architectures - curtis chambers." Video `https://www.youtube.com/watch?v=Jups7FveC1E`, Dec. 2011. Accessed: 2017-12-01.

[53] "NPM." `npmjs.com`. Accessed 2017-12-01.

[54] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*. ECMA, 5.1 ed., June 2011.

[55] A. H. Borning, "Classes versus prototypes in object-oriented languages," in *Proceedings of 1986 ACM Fall joint computer conference*, pp. 36–40, IEEE Computer Society Press, 1986.

[56] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of JavaScript," in *European Conference on Object-Oriented Programming*, pp. 435–458, Springer, 2012.

[57] F. Logozzo and H. Venter, "Rata: rapid atomic type analysis by abstract interpretation–application to JavaScript optimization," in *International Conference on Compiler Construction*, pp. 66–83, Springer, 2010.

[58] S. Wei and B. G. Ryder, "Practical blended taint analysis for JavaScript," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 336–346, ACM, 2013.

[59] G. M. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pp. 257–281, 2014.

[60] G. Richards, F. Zappa Nardelli, and J. Vitek, "Concrete types for TypeScript," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 37, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[61] A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris, "Safe & efficient gradual typing for TypeScript," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pp. 167–180, 2015.

[62] P. Gardner, S. Maffeis, and G. D. Smith, "Towards a program logic for JavaScript," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pp. 31–44, 2012.

[63] D. Park, A. Stefanescu, and G. Rosu, "KJS: a complete formal semantics of javascript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pp. 346–356, 2015.

[64] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of JavaScript," in *European conference on Object-oriented programming*, pp. 126–150, Springer, 2010.

[65] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[66] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pp. 1–10, IEEE, 2010.

[67] L. O. Ergin, "The story behind anyone can login as root tweet." `https://medium.com/@lemiorhan/the-story-behind-anyone-can-login-as-root-tweet-33731b5ded71`, Nov. 2017. Accessed: 2018-03-25.

[68] B. Schneier, “Debating full disclosure.” https://www.schneier.com/blog/archives/2007/01/debating_full_d.html, Jan. 2007. Accessed: 2018-03-25.

[69] T. Hunt, “The responsibility of public disclosure.” https://www.troyhunt.com/the-responsibility-of-public-disclosure/, May 2013. Accessed: 2018-03-27.

[70] T. Hunt, “Kids Pass just reminded us how hard responsible disclosure is.” https://www.troyhunt.com/kids-pass-just-reminded-us-how-hard-responsible-disclosure-is/, July 2017. Accessed: 2018-03-25.

[71] Google Project Zero, “Feedback and data-driven updates to Googles disclosure policy.” https://googleprojectzero.blogspot.co.uk/2015/02/feedback-and-data-driven-updates-to.html, Feb. 2015. Accessed: 2018-03-25.

Chapter 8: **Appendix**

8.1 Professional Issues

It is possible to imagine a world in which ExpoSE is deployed as a tool to automatically test open source code from GitHub or npm, either as part of a survey of open source JavaScript software or as a free service. Similar tools already exist such as Snyk, which attempts to find dependencies with known vulnerabilities, and the static analysis tool Deepscan, which attempts to find common coding errors. In such a situation it would be important to consider how the process of disclosing a a bug would work, given that ExpoSE has no framework for identifying the severity or impact of the bugs it finds, and the difficulty in doing so reliably [66]. Whilst ExpoSE cannot produce false positives, the impact of a bug found by it can range from a low impact bug that is unlikely to be triggered to a serious security vulnerability.

The steps to take to disclose a flaw ethically are not universally agreed upon and existing literature mainly deals with security vulnerabilities due to their critical nature. There are two broad types of disclosure that could qualify as ethical. The first is *full disclosure*, in which the bug is reported as early and widely as possible with the intent of informing end users about the flaw. The second is *responsible disclosure*, in which the flaw is reported to the maintainer or vendor and no one else until the flaw is rectified. Whilst the definition of full disclosure is concrete, the definition of responsible disclosure is more ambiguous and provokes a number of questions. What do you do if the flaw is not rectified? At what point does the risk of potential harm require you to release the details of the vulnerability to the public? Does the amount of time given vary with the severity of the bug? These questions, and the discussion about the best method for disclosure, are still active areas of debate today. This debate, and the issues with the boilerplate definition of responsible disclosure, is highlighted by two high profile vulnerabilities disclosed this year.

The Meltdown and Spectre vulnerabilities, which revealed underlying flaws in the methods for speculative execution in CPUs across multiple vendors, were announced in January. Given the nature of the vulnerability numerous parties were required to know about the flaw, including operating system and browser vendors as well as cloud computing platforms such as AWS and Azure. The additional complexity created by the number of groups involved resulted in a chaotic public disclosure due to a broken embargo. The number of parties involved also had a drastic effect on the timeline of discovery to disclosure. Google's Project Zero, the team behind the initial discovery, have a standard 90 day window after which they disclose but, due to the complexity of the vulnerability and the number of groups involved, this was doubled to 6 months.

Whilst the Meltdown and Spectre vulnerabilities are clearly extraordinary circumstances they highlight one of the issues with the standard model of responsible disclosure - a long period where users are unknowingly affected and vulnerable to exploitation.

Another high profile vulnerability disclosed this year was the High Sierra Mac OS login bypass which enabled users to access root privileges without a password. The bug was discovered by Lemi Ergin and was disclosed via Twitter leading him to face criticism for failing to follow responsible disclosure principles. Whilst it is not considered good practise to not work with the vendor when disclosing a vulnerability, Ergin's tweet prompted Apple to release a next day update whilst Ergin's company's email earlier in the week and a post on Apple's Developer Forum failed to prompt action [67].

Whilst most professionals would broadly agree that responsible disclosure is the preferred method, there are respected individuals within the security field such as Bruce Schneier that argue in favour of full disclosure [68]. He argues that "public scrutiny is how security improves" and that it is important that consumers have "all the information" to "make an informed decision about security".

I think it is worth re-evaluating the loaded term of responsible disclosure and examining what, if any, role full disclosure has to play. It is clear from the Ergin's tweet that full disclosure produces results, but at what risk? A common argument made in favour of responsible disclosure is that sharing the details of a vulnerability is irresponsible - that the 'bad guys' can abuse them if they're made public. Schneier argues that malicious users are often more advanced than security researchers and may already know about the vulnerability and be actively exploiting it. In addition, if malciious actors are not aware of the vulnerability - Schneider suggests that releasing the details of a vulnerability is a reasonable trade-off for the increased visibility of the risk that users have. Conversely security professional Troy Hunt, creator of the data breach service `of have i been pwned?`, argues that public disclosure is not appropriate if it would immediately increase the risk to users [69]. However, in cases where responsible disclosure is made difficult by unwilling disclosure recipients (and in cases with organisations and individuals who have a poor track record of responding to disclosures), Hunt believes it is in the public's interest to perform full disclosure [70].

Google's Project Zero, the security research team that initially identified the Spectre and Meltdown vulnerabilities, have also struggled to identify the best way to handle disclosure. They received criticism for their strict handling of a vulnerability that they discovered in Windows 7 which was publicly disclosed two days before a patch was released by Microsoft. As a result of the backlash Project Zero added a 14 day grace period to their 90 day disclosure policy but they also make provision for certain "extreme circumstances, in which they reserve the right to bring the deadline forward or backwards [71]. In doing so they acknowledge the complexity of disclosing security vulnerabilities.

In conclusion, when considering how to responsibly disclose I believe there are a number of factors that must be considered: the risk to users, the number of organisations that need to be involved, the vendor's history of compliance, and if there are known exploits for the vulnerability already being used. I reject the notion that there is a 'one size fits all' approach to responsible disclosure and suggest that, whilst full disclosure should not be the default course of action, it warrants consideration in situations when disclosure would present no risk to users or when dealing with organisations or individuals who have a history of non-compliance.

My conclusion prompts the question: how can responsible disclosure be achieved in the hypothetical world in which ExpoSE is deployed as an automated tool to discover bugs? I believe that every bug found would have to be treated as a possible vulnerability and be reported privately to the maintainer. I would provide a 90 day window for response, after which I would post the vulnerability to a public issue tracker and attempt to alert any public codebases which depended on that code.

8.2 Manual

8.2.1 Installation

ExpoSE is a Node.js program and as such requires `node` and `npm` to be installed. It is recommended to use version 8.10.0.

The easiest way of installing the recommended `node` version is using `nvm` which can be achieved by running the commands below and following any directions given. ExpoSE's installation process also requires `clang`, `clang++`, `gnuplot`, `make`, and `python2`.

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.8/install.sh | bash
nvm install 8.10.0
```

To install node navigate to the project directory and type `npm start`. This process may take a while as ExpoSE compiles Z3 from source.

8.2.2 Testing Programs

ExpoSE is invoked from the command line through the `expoSE` script, which is installed to the path on first run.

ExpoSE extends JavaScript syntax with the additional keyword `symbolic`, which when used `var x = symbolic MyFirstSymbolicVariable;` marks the variable as a pure symbolic variable – a symbolic variable which can be of any type. Symbolic variables can also be created with an initial value which fixes their type `var x = symbolic X initial [1, 2, 3];`.

Programs do not need to be rewritten if they have public functions which can be explored as ExpoSE has an automatic harness generator which is invoked with `expoSE ahg myTarget.js`.

ExpoSE also has a number of important environment variables. `EXPOSE_PRINT_COVERAGE=1` prints out the program under test and indicates which lines are covered. `EXPOSE_PRINT_PATHS=1` prints the output of each test case. `EXPOSE_LOG_LEVEL` sets the log level between 0 (silent) and 3 (noisy) - I would set this no higher than 2. `NO_COMPILE=1` stops ExpoSE from recompiling after each execution - use this!

8.2.3 Testing Arrays

There are a number of array specific tests in `tests/arrays/`. They can be all run at once using `scripts/array_tests` or they can be run individually like `EXPOSE_PRINT_PATHS=1 EXPOSE_LOG_LEVEL =2 expoSE tests/arrays/array_index_get_or.js`.

Useful tests to run as micro programs are:

- `array_length_numbers.js`
- `array_index_matches_non_symbolic_value.js`
- `array_index_matches_symbolic_value.js`
- `array_index_getter_is_symbolic.js`