# A Symbolic Execution Framework for JavaScript

*Prateek Saxena*
*Devdatta Akhawe*
*Steve Hanna*
*Feng Mao*
*Stephen McCamant*
*Dawn Song*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-26
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-26.html

March 8, 2010

# A Symbolic Execution Framework for JavaScript

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song
*Computer Science Division, EECS Department*
*University of California, Berkeley*
{*prateeks, devdatta, sch, fmao, smcc, dawnsong*}*@cs.berkeley.edu*

***Abstract*—As AJAX applications gain popularity, client-side JavaScript code is becoming increasingly complex. However, few automated vulnerability analysis tools for JavaScript exist. In this paper, we describe the first system for exploring the execution space of JavaScript code using symbolic execution. To handle JavaScript code's complex use of string operations, we design a new language of string constraints and implement a solver for it. We build an automatic end-to-end tool, Kudzu, and apply it to the problem of finding client-side code injection vulnerabilities. In experiments on 18 live web applications, Kudzu automatically discovers 2 previously unknown vulnerabilities and 9 more that were previously found only with a manually-constructed test suite.**

***Keywords*-web security; symbolic execution; string decision procedures**

## I. Introduction

Rich web applications have a significant fraction of their code written in client-side scripting languages, such as JavaScript. As an increasing fraction of code is found on the client, client-side security vulnerabilities (such as client-side code injection [19], [25]–[27]) are becoming a prominent threat. However, a majority of the research on web vulnerabilities so far has focused on server-side application code written in PHP and Java. There is a growing need for powerful analysis tools for the client-side components of web applications. This paper presents the first techniques and system for automatically exploring the execution space of client-side JavaScript code. To explore this execution space, our techniques generate new inputs to cover a program's *value space* using dynamic symbolic execution of JavaScript, and to cover its *event space* by automatic GUI exploration.

Dynamic symbolic execution for JavaScript has numerous applications in web security. In this paper we focus on one of these applications: automatically finding client-side code injection vulnerabilities. A client-side code injection attack occurs when client-side code passes untrusted input to a dynamic code evaluation construct, without proper validation or sanitization, allowing an attacker to inject JavaScript code that runs with the privileges of a web application.

JavaScript execution space exploration is challenging for many reasons. In particular, JavaScript applications accept many kinds of input, and those inputs are structured just as strings. For instance, a typical application might take user input from form fields, messages from its server via `XMLHttpRequest`, and data from code running concurrently in other browser windows. Each kind of input string has its own format, so developers use a combination of custom routines and third-party libraries to parse and validate the inputs they receive. To effectively explore a program's execution space, a tool must be able to supply values for all of these different kinds of inputs and reason about how they are parsed and validated.

**Approach.** In this paper, we develop the first complete symbolic-execution based framework for client-side JavaScript code analysis. We build an automated, standalone tool that, given a URL for a web application, automatically generates high-coverage test cases to systematically explore its execution space. Automatically reasoning about the operations we see in real JavaScript applications requires a powerful constraint solver, especially for the theory of strings. However, the power needed to express the semantics of JavaScript operations is beyond what existing string constraint solvers [14], [17] offer. As a central contribution of this work, we overcome this difficulty by proposing a constraint language and building a practical solver that supports the specification of boolean, machine integer (bit-vector), and string constraints, including regular expressions, over multiple variable-length string inputs. This language's rich support for string operations is crucial for reasoning about the parsing and validation checks that JavaScript applications perform.

To show the practicality of our constraint language, we detail a translation from the most commonly used JavaScript string operations to our constraints. This translation also harnesses concrete information from a dynamic execution of the program in a way that allows the analysis to scale. We analyze the theoretical expressiveness of the theory of strings supported by our language (including in comparison to existing constraint solvers), and bound its computational complexity. We then give a sound and complete decision procedure for the bounded-length version of the constraint language. We develop an end-to-end system, called *Kudzu*, that performs symbolic execution with this constraint solver at its core.

**End-to-end system.** We identify further challenges in building an end-to-end automated tool for rich web applications. For instance, because JavaScript code interacts closely with a

user interface, its input space can be divided into two classes, the *events space* and the *value space*. The former includes the state (check boxes, list selections) and sequence of actions of user-interface elements, while the latter includes the contents of external inputs. These kinds of input jointly determine the code's behavior, but they are suited to different exploration techniques. Kudzu uses GUI exploration to explore the event space, and symbolic execution to explore the value space.

We evaluate Kudzu's end-to-end effectiveness by applying it to a collection of 18 JavaScript applications. The results show that Kudzu is effective at getting good coverage by discovering new execution paths, and it automatically discovers 2 previously-unknown vulnerabilities, as well as 9 client-side code injection vulnerabilities that were previously found only with a manually-created test suite.

**Contributions.** In summary, this paper makes the following main contributions:

- We identify the limitations of previous string constraint languages that make them insufficient for parsing-heavy JavaScript code, and design a new constraint language to resolve those limitations. (Section IV)
- We design and implement a practical decision procedure for this constraint language. (Section V)
- We build the first symbolic execution engine for JavaScript, using our constraint solver. (Sections III and VI)
- Combining symbolic execution of JavaScript with automatic GUI exploration and other needed components, we build the first end-to-end automated system for exploration of client-side JavaScript. (Section III)
- We demonstrate the practical use of our implementation by applying it to automatically discovering 11 client-side code injection vulnerabilities, including two that were previously unknown. (Section VII)

## II. Problem Statement and Overview

In this section we state the problem we focus on, exploring the execution space of JavaScript applications; describe one of its applications, finding client-side code injection vulnerabilities; and give an overview of our approach.

**Problem statement.** We develop techniques to systematically explore the execution space of JavaScript application code.

JavaScript applications often take many kinds of input. We view the input space of a JavaScript program as split into two categories: the *event space* and the *value space*.

- *Event space*. Rich web applications typically define tens to hundreds of JavaScript event handlers, which may execute in any order as a result of user actions such as clicking buttons or submitting forms. Event handler code may check the state of GUI elements (such as check-boxes or selection lists). The ordering of events and the state of the GUI elements together affects the behavior of the application code.
- *Value space*. The values of inputs supplied to a program also determine its behavior. JavaScript has numerous interfaces through which input is received:
  - *User data*. Form fields, text areas, and so on.
  - *URL and cross-window communication abstractions*. Web principals hosted in other windows or frames can communicate with JavaScript code via inter-frame communication abstractions such as URL fragment identifiers and HTML 5's proposed `postMessage`, or via URL parameters.
  - *HTTP channels*. Client-side JavaScript code can exchange data with its originating web server using `XMLHttpRequest`, HTTP cookies, or additional HTTP `GET` or `POST` requests.

This paper primarily focuses on techniques to systematically explore the value space using symbolic execution of JavaScript, with the goal of generating inputs that exercise new program paths. However, automatically exploring the event space is also required to achieve good coverage. To demonstrate the efficacy of our techniques in an end-to-end system, we combine symbolic execution of JavaScript for the value space with a GUI exploration technique for the event space. This full system is able to automatically explore the combined input space of client-side web application code.

**Application: finding client-side code injection vulnerabilities.** Exploring a program's execution space has a number of applications in the security of client-side web applications. In this paper, we focus specifically on one security application, finding client-side code injection vulnerabilities.

Client-side code injection attacks, which are sometimes referred to as DOM-based XSS, occur when client-side code uses untrusted input data in dynamic code evaluation constructs without sufficient validation. Like reflected or stored XSS attacks, client-side code injection vulnerabilities can be used to inject script code chosen by an attacker, giving the attacker the full privileges of the web application. We call the program input that supplies the data for an attack the *untrusted source*, and the potentially vulnerable code evaluation construct the *critical sink*. Examples of critical sinks include `eval`, and HTML creation interfaces like `document.write` and `.innerHTML`.

In our threat model, we treat all URLs and cross-window communication abstractions as untrusted sources, as such inputs may be controlled by an untrusted web principal. In addition, we also treat user data as an untrusted source because we aim to find cases where user data may be interpreted as code. The severity of attacks from user-data on client-side is often less severe than a remote XSS attack, but developers tend to fix these and Kudzu takes a conservative approach of reporting them. HTTP channels such as `XMLHttpRequest` are currently restricted to communicating with a web server

from the same domain as the client application, so we do not treat them as untrusted sources. Developers may wish to treat HTTP channels as untrusted in the future when determining susceptibility to cross-channel scripting attacks [5], or when enhanced abstractions (such as the proposed cross-origin XMLHttpRequest [29]) allow cross-domain HTTP communication directly from JavaScript.

To effectively find XSS vulnerabilities, we require two capabilities: (a) generating directed test cases that explore the execution space of the program, and (b) checking, on a given execution path, whether the program validates all untrusted data sufficiently before using it in a critical sink. Custom validation checks and parsing routines are the norm rather than the exception in JavaScript applications, so our tool must check the behavior of validation rather than simply confirming that it is performed.

In previous work, we developed a tool called FLAX which employs taint-guided fuzzing for finding client-side code injection attacks [26]. However, FLAX relies on an external, manually developed test harness to explore the path space. Kudzu, in contrast, automatically generates a test suite that explores the execution space systematically. Kudzu also uses symbolic reasoning (with its constraint solver) to check if the validation logic employed by the application is sufficient to block malicious inputs — this is a one-step mechanism for directed exploit generation as opposed to multiple rounds of undirected fuzzing employed in FLAX. Static analysis techniques have also been employed for JavaScript [12] to reason about multiple paths, but can suffer from false positives and do not produce test inputs or attack instances. Symbolic analyses and model-checking have been used for server-side code [2], [20]; however, the complexity of path conditions we observe requires more expressive symbolic reasoning than supported by tools for server-side code.

**Approach Overview.** The value space and event space of a web application are two different components of its input space: code reachable by exploring one part of the input space may not be reachable by exploring the other component alone. For instance, exploring the GUI event space results in discovering new views of the web application, but this does not directly affect the coverage that can be achieved by systematically exploring all the paths in the code implementing each view. Conversely, maximizing path coverage is unlikely to discover functionality of the application that only happens when the user explores a different application view. Therefore, Kudzu employs different techniques to explore each part of the input space independently.

*Value space exploration*. To systematically explore different execution paths, we develop a component that performs dynamic symbolic execution of JavaScript code, and a new constraint solver that offers the desired expressiveness for automatic symbolic reasoning.

In dynamic symbolic execution, certain inputs are treated as symbolic variables. Dynamic symbolic execution differs from normal execution in that while many variable have their usual (*concrete*) values, like 5 for an integer variable, the values of other variables which depend on symbolic inputs are represented by *symbolic* formulas over the symbolic inputs, like $input_1 + 5$. Whenever any of the operands of a JavaScript operation is symbolic, the operation is simulated by creating a formula for the result of the operation in terms of the formulas for the operands. When a symbolic value propagates to the condition of a branch, Kudzu can use its constraint solver to search for an input to the program that would cause the branch to make the opposite choice.

*Event space exploration*. As a component of Kudzu we develop a GUI explorer that searches the space of all event sequences using a random exploration strategy. Kudzu's GUI explorer component randomly selects an ordering among the user events registered by the web page, and automatically fires these events using an instrumented version of the web browser. Kudzu also has an input-feedback component that can replay the sequence of GUI events explored in any given run, along with feeding new values generated by the constraint solver to the application's data inputs.

*Testing for client-side code injection vulnerabilities*. For each input explored, Kudzu determines whether there is a flow of data from an untrusted data source to a critical sink. If it finds one, it seeks to determine whether the program sanitizes and/or validates the input correctly to prevent attackers from injecting dangerous elements into the critical sink. Specifically, it attempts to prove that the validation is insufficient by constructing an attack input. As we will describe in more detail in Section III-B, it combines the results of symbolic execution with a specification for attacks to create a constraint solver query. If the constraint solver finds a solution to the query, it represents an attack that can reach the critical sink and exploit a client-side code injection vulnerability.

## III. End-to-End System Design

This section describes the various components that work together to make a complete Kudzu-based vulnerability-discovery system work. The full explanation of the constraint solver is in Sections IV through VI. For reference, the relationships between the components are summarized in Figure 1.

### A. System Components

First, we discuss the core components that would be used in any application of Kudzu: the *GUI explorer* that generates input events to explore the event space, the *dynamic symbolic interpreter* that performs symbolic execution of JavaScript, the *path constraint extractor* that builds queries based on the results of symbolic execution, the *constraint solver* that finds satisfying assignments to those queries, and the *input*

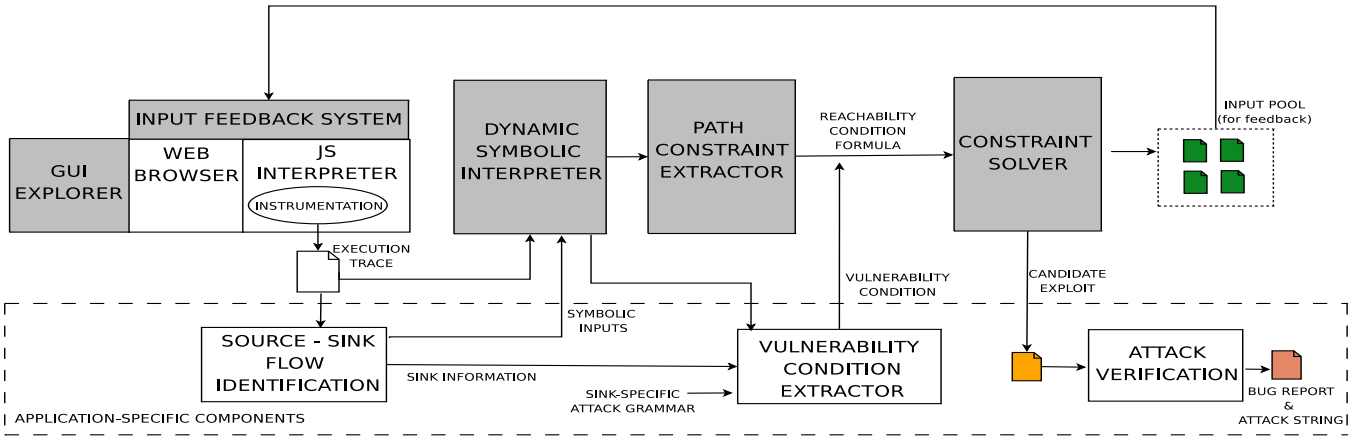


Figure 1: Architecture diagram for Kudzu. The components drawn in the dashed box perform functions specific to our application of finding client-side code injection. The remaining components are application-agnostic. Components shaded in light gray are the core contribution of this paper.

*feedback* component that uses the results from the constraint solver as new program inputs.

**The GUI explorer.** The first step in automating JavaScript application analysis is exploring the event space of user interactions. Each event corresponds to a user interaction such as clicking a check-box or a button, setting focus on a field, adding data to data fields, clicking a link, and so on. Kudzu currently explores the space of all sequences of events using a random exploration strategy. One of the challenges is to comprehensively detect all events that could result in JavaScript code execution. To address this, Kudzu instruments the browser functions that process HTML elements on the current web page to record when an event handler is created or destroyed. Kudzu's GUI explorer component randomly selects an ordering among the user events registered by the web page and executes them[1]. The random seed can be controlled to replay the same ordering of events. While invoking handlers, the GUI component also generates (benign) random test strings to fill text fields. (Later, symbolic execution will generate new input values for these fields to explore the input space further.) Links that navigate the page away from the application's domain are cancelled, thereby constraining the testing to a single application domain at a time. In the future, we plan to investigate alternative strategies to prioritize the execution of events discovered as well.

**Dynamic symbolic interpreter.** Kudzu performs dynamic symbolic execution by first recording an execution of the program with concrete inputs, and then symbolically interpreting the recorded execution in a dynamic symbolic interpreter. For recording an execution trace, Kudzu employs an existing instrumentation component [26] implemented in the web browser's JavaScript interpreter. For each JavaScript bytecode instruction executed, it records the semantics of the operation, its operands and operand values in a simplified intermediate language called JASIL [26]. The set of JavaScript operations captured includes all operations on integers, booleans, strings, arrays, as well as control-flow decisions, object types, and calls to browser-native methods. For the second step, dynamic symbolic execution, we have developed from scratch a symbolic interpreter for the recorded JASIL instructions.

Symbolic inputs for Kudzu are configurable to match the needs of an application. For instance, in the application we consider, detecting client-side code injection, all URL data, data received over cross-window communication abstractions, and user data fields are marked symbolic. Symbolic inputs may be strings, integers, or booleans. Symbolic execution proceeds on the JASIL instructions in the order they are recorded in the execution trace. At any point during dynamic symbolic execution, a given operand is either *symbolic* or *concrete*. If the operand is symbolic, it is associated with a *symbolic value*; otherwise, its value is purely concrete and is stored in the dynamic execution trace. When interpreting a JASIL operation in the dynamic symbolic interpreter, the operation is symbolically executed if one or more of its input operands is symbolic. Otherwise the operation of the symbolic interpreter on concrete values would be exactly the same as the real JavaScript interpreter, so we simply reuse the concrete results already stored in the execution trace.

The symbolic value of an operand is a formula that represents its computation from the symbolic inputs. For instance, for the JASIL assignment operation `y := x`, if `x` is symbolic (say, with the value $input_1 + 5$), then symbolic execution of the operation copies this value to `y`, giving

[1]Invoking an event handler may invalidate another handler (for instance, when the page navigates as a result). In that case, the invalidated handlers are ignored and if new handlers are created by the event that causes invalidation, these events are explored subsequently.

y the same symbolic value. For an arithmetic operation, say y := x1 + x2 where x1 is symbolic (say with value $input_2 + 3$) and x2 is not (say with the concrete value 7), the symbolic value for y is the formula representing the sum ($input_2 + 10$). Operations over strings and booleans are treated in the same way, generating formulas that involve string operations like match and boolean operations like and. At this point, string operations are treated simply as uninterpreted functions. During the symbolic execution, whenever the symbolic interpreter encounters an operation outside the supported formula grammar, it forces the destination operand to be concrete. For instance, if the operation is x = parseFloat(s) for a symbolic string s, x and s can be replaced with their concrete values from the trace (say, 4.3 and "4.3"). This allows symbolic computation to continue for other values in the execution.

**Path constraint extractor.** The execution trace records each control-flow branch (e.g., if statement) encountered during execution, along with the concrete value (true or false) representing whether the branch was taken. During symbolic execution, the corresponding *branch condition* is recorded by the path constraint extractor if it is symbolic. As execution continues, the formula formed by conjoining the symbolic branch conditions (negating the conditions of branches that were not taken) is called the *path constraint*. If an input value satisfies the path constraint, then an execution of the program on that input will follow the same execution path.

To explore a different execution path, Kudzu selects a branch on the execution path and builds a modified path constraint that is the same up to that branch, but that has the negation of that branch condition (later conditions from the original branch are omitted). An input that satisfies this condition will execute along the same path up to the selected branch, and then explore the opposite alternative. There are several strategies for picking the order in which branch conditions can be negated — Kudzu currently uses a generational search strategy [11].

**Constraint solver.** Most symbolic execution tools in the past have relied on an existing constraint solver. However, Kudzu generates a rich set of constraints over string, integer and boolean variables for which existing off-the-shelf solvers are not powerful enough. Therefore, we have built a new solver for our constraints (we present the algorithm and design details in Section V). In designing this component, we examined the symbolic constraints Kudzu generates in practice. From the string constraints arising in these, we distilled a set of primitive operations required in a core constraint language. (This core language is detailed in Section IV, while the solver's full interface is given in Section VI.) We justify our intuition that solving the core constraints is sufficient to model JavaScript string operations in Section VI, where we show a practical translation of JavaScript string operations into our constraint language.

**Input feedback.** Solving the path constraint formula using the solver creates a new input that explores a new program path. These newly generated inputs must be fed back to the JavaScript program: for instance simulated user inputs must go in their text fields, and GUI events should be replayed in the same sequence as on the original run. The input feedback component is responsible for this task. As a particular HTML element (e.g a text field) in a document is likely allocated a different memory address on every execution, the input feedback component uses XPath [31] and DOM identifiers to uniquely identify HTML elements across executions and feed appropriate values into them. If an input comes from an attribute for a DOM object, the input feedback component sets that attribute when the object is created. If the input comes via a property of an event that is generated by the browser when handling cross-window communication, such as the origin and data properties of a postMessage event, the component updates that property when the JavaScript engine accesses it. Kudzu instruments the web browser to determine the context of accesses, to distinguish between accesses coming from the JavaScript engine and accesses coming from the browser core or instrumentation code.

### B. Application-specific components

Next, we discuss three components that are specialized for the task of finding client-side code injection vulnerabilities: a *sink-source identification* component that determines which critical sinks might receive untrusted input, a *vulnerability condition extractor* that captures domain knowledge about client-side code injection attacks, and the *attack verification* component that checks whether inputs generated by the tool in fact represent exploits.

**Sink-source identification.** To identify if external inputs are used in critical sink operations such as eval or document.write, we perform a dynamic data flow analysis on the execution trace. As outlined earlier, we treat all URL data, data received over cross-window communication abstractions (such as postMessage), and data filled into user data fields as potentially untrusted. The data flow analysis is similar to a dynamic taint analysis. Any execution trace that reveals a flow of data to a critical sink is subject to further symbolic analysis for exploit generation. We use an existing framework, FLAX, for this instrumentation and taint-tracking [26] in a manner that is faithful to the implementation of JavaScript in the WebKit interpreter.

**Vulnerability condition extractor.** An input represents an attack against a program if it passes the program's validation checks, but nonetheless implements the attacker's goals (i.e., causes a client-side code injection attack) when it reaches a critical sink. The vulnerability condition extractor collects from the symbolic interpreter a formula representing the

(possibly transformed) value used at a critical sink, and combines it with the path constraint to create a formula describing the program's validation of the input.[2]. To determine whether this value constitutes an attack, the vulnerability condition extractor applies a sink-specific vulnerability condition specification, which is a (regular) grammar encoding a set of strings that would constitute an attack against a particular sink. This specification is conjoined with the formula representing the transformed input to create a formula representing values that are still dangerous after the transformation.

For instance, in the case of the `eval` sink, the vulnerability specification asserts that a valid attack should be zero or more statements each terminated by a ';', followed by the payload. Such grammars can be constructed by using publicly available attack patterns [13]. The tool's attack grammars are currently simple and can be extended easily for comprehensiveness and to incorporate new attacks.

To search only for realistic attacks, the specification also incorporates domain knowledge about the possible values of certain inputs. For instance, when a string variable corresponds to the web URL for the application, we assert that the string starts with the same domain as the application.

**Attack verification.** Kudzu automatically tests the exploit instance by feeding the input back to the application, and checking if the attack payload (such as a script with an alert message) is executed. If this verification fails, Kudzu does not report an alarm.

## IV. Core Constraint Language

In order to support a rich language of input constraints with a simple solving back end, we have designed an intermediate form we call the *core constraint language*. This language is rich enough to express constraints from JavaScript programs, but simple enough to make solving the constraints efficient. In this section we define the constraint language, analyze its expressiveness and the theoretical complexity of deciding it, and compare its expressiveness to the core languages of previous solvers.

### A. Language Definition

The abstract syntax for our core constraint language is shown in Figure 2. A formula in the language is an arbitrary boolean combination of constraints. Variables which represent strings may appear in five types of constraints. The first three constraint types indicate that a string is a member of the language defined by a regular expression, that two strings are equal, or one string is equal to the concatenation of two other strings. The two remaining constraints relate

[2]Validation and sanitization for critical client-side sink operations may happen on the server side (when data is sent back via `XMLHttpRequest`). Our implementation deals with this by recognizing such transformations using approximate tainting techniques [26] across the data sent and received over XMLHttpRequests.

$$
\begin{array}{rcl}
\mathit{Formula} & ::= & \neg \mathit{Formula} \\
 & | & \mathit{Formula} \wedge \mathit{Formula} \\
 & | & \mathit{Constraint} \\
\mathit{Constraint} & ::= & \mathit{Var} \in \mathit{RegExp} \\
 & | & \mathit{Var} = \mathit{Var} \\
 & | & \mathit{Var} = \mathit{Var} \circ \mathit{Var} \\
 & | & \texttt{length}(\mathit{Var})\ \mathit{Rel}\ \mathit{Number} \\
 & | & \texttt{length}(\mathit{Var})\ \mathit{Rel}\ \texttt{length}(\mathit{Var}) \\
\mathit{RegExp} & ::= & \mathit{Character} \\
 & | & \epsilon \\
 & | & \mathit{RegExp}\ \mathit{RegExp} \\
 & | & \mathit{RegExp} \mid \mathit{RegExp} \\
 & | & \mathit{RexExp}^{\star} \\
\mathit{Rel} & ::= & < \mid \leq \mid = \mid \geq \mid >
\end{array}
$$

Figure 2: Abstract grammar of the core constraint language.

the length of one string to a constant natural number, or to the length of another string, by any of the usual equality or ordering operations. Regular expressions are formed from characters or the empty string (denoted by $\epsilon$) via the usual operations of concatenation (represented by juxtaposition), alternation ($|$), and repetition zero or more times (Kleene star $\star$).

The constraints all have their usual meanings. Any number of variables may be introduced, and *Character*s are drawn from an arbitrary non-empty alphabet, but *Number*s must be non-negative integers. For present purposes, strings may be of unbounded length, though we will introduce upper bounds on their lengths later.

### B. Expressiveness and Complexity

Though the core constraint language is intentionally small, it is not minimal: some types of constraints are included for the convenience of translating to and from the core language, but do not fundamentally increase its expressiveness. String equality, comparisons between lengths and constants, and inequality comparisons between lengths can be expressed using concatenation, regular expressions, and equality between string lengths respectively; the details are omitted for space.

Each of the remaining constraint types (regular expression membership, concatenation, and length) makes its own contribution to the expressiveness of the core constraints. Appendix A gives examples of the sets of strings that each constraint type can uniquely define. The core constraint language is expressive enough that the complexity of deciding it is not known precisely; it is at least PSPACE-hard. These relationships are summarized in Figure 3. The complexity of our core constraint language falls to NP-complete when the lengths of string variables are bounded, as they are in our implementation. Further details are in Appendix B.

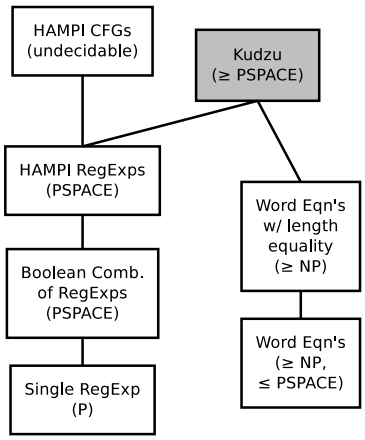


Figure 3: Relations between the unbounded versions of several theories of strings. Theories higher in the graph are strictly more expressive but are also at least as complex to decide. Kudzu's core constraint language (shaded) is strictly more expressive than either the core language of HAMPI [17] or the theory of word equations and an equal length predicate (the "pure library language" of [4]).

### C. Expressiveness Comparison

Our system's core constraint language is more expressive than the constraints used in similar previous systems, and this expressiveness is key in allowing it to handle a more complex class of applications.

Bjørner *et al.* [4] present a "pure library language" that, like our core constraint language, includes word equations and the ability to assert that two strings have the same length, so like our language its decidability is open. However, their language does not include regular expressions. Regular expressions may be less common in the .NET applications Bjørner *et al.* study, but they are used ubiquitously in JavaScript, so regular expression support is mandatory in our domain. Similarly the work of Caballero *et al.* [3], [7] deals with programs compiled from C-family languages, whose string operations are much more limited.

The DPRLE tool [14] focuses on a class of constraints that combine concatenation and regular expression matching, but in a more limited way than our tool supports. DPRLE addresses a different problem domain, since it gives solutions for constraints over languages (potentially infinite sets of strings) rather than single strings, but this makes the task significantly more difficult. We were unable to express the constraints from our application in DPRLE's input format or any straightforward extension of it. For instance, there is no way to express the constraint that two language expressions should be equal, not surprising since such constraints in general are undecidable [8].

HAMPI [17] provides support for regular expression constraints (and in fact we build on its implementation for this feature), but its support for other constraints is limited, particularly by the fact that it supports only a single string variable. The variable can be concatenated with constant strings, but these string expressions cannot be compared with each other, only with regular expressions, so HAMPI lacks the full generality of word equations. For instance, HAMPI constraints cannot define the set $\{uv\#u\#v : u, v \in \{0,1\}^*\}$.

It is worth reemphasizing that these limitations are not just theoretical: they make these previous systems unsuitable for our applications. One of the most common operations in the programs we examine is to parse a single input string (such as a URL) into separate input variables using `split` or repeated regular expression matching. Representing the semantics of such an operation requires relating the contents of one string variable to those of another, something that neither DPRLE nor HAMPI supports.

## V. Core Constraint Solving Approach

In this section, we explain our algorithm for solving the core set of constraints. We introduce a bounded version of the constraints where we assume a user-supplied upper bound $k$ on the length of the variables. This allows us to employ a SAT-based solution strategy without reducing the practical expressiveness of the language.

The algorithm satisfies three important properties, whose proof appears in Appendix C:

1) *Soundness*. Whenever the algorithm terminates with an assignment of values to string variables, the solution is consistent with the input constraints.
2) *Bound-k completeness*. If there exists a solution for the string variables where all strings have length $k$ or less, then our algorithm finds one such solution.
3) *Termination*. The algorithm requires only a finite number of steps (a function of the bound) for any input.

The solver translates constraints on the contents of strings into bit-vector constraints that can be solved with a SAT-based SMT solver. For this purpose, the solver translates each input string into a sequence of $n$-bit integers ($n = 8$ in the current implementation). Each string variable $S$ also has an associated integer variable $L_S$ representing its length. A single string is converted to a bit-vector by concatenating the binary representations of each character. Then, the bit-vectors representing each string are themselves concatenated into a single long bit-vector. (The order in which the strings are concatenated into the long vector reflects the concatenation constraints, as detailed in step 1 below.) The solver passes the constraints over this bit vector to a SMT (satisfiability modulo theories) decision procedure for the theory of bit vectors, STP [10] in our implementation. Informally, it is convenient to refer to the combined bit vector as if it were an array indexed by character offsets, but we do not use STP's theory of arrays, and character offsets are multiplied by $n$ to give bit offsets before producing the final constraints.

```
Input: C : constraint list
Output: (IsSat : bool, Solutions : string list)
G ← BuildConcatGraph(C);
(C′, StrOrderMap) ← DecideOrder(G);
C ← C ∪ C′;
FailLenDB : length_assignment list ← ∅;
while true do
    (X, lengths) ← SolveLengths(C, FailLenDB);
    if (X = UNSAT) then
        print "Unsatisfiable";
        halt(false, ∅);
    end
    Final : bitvector_constraints;
    Final ← CreateBVConstraints(StrOrderMap, C, lengths);
    (Result, BVSolutions) ← BVSolver(Final);
    if (Result = SAT) then
        print "Satisfiable";
        printSolutions(BVSolutions, lengths, StrOrderMap);
        halt(true, BVsToStrings(BVSolutions));
    end
    else
        FailLenDB ← FailLenDB ∪ lengths;
    end
end
```

Figure 4: Algorithm for solving the core constraints.

Our algorithm is shown in Figure 4. At a high level, it has three steps. First, it translates string concatenation constraints into a layout of variables (with overlap) in the final character array mentioned above. Second, it extracts integer constraints on the lengths of strings and finds a satisfying length assignment using the SMT solver. Finally, given a position and length for each string, the solver translates the constraints on the contents of each string into bit-vector constraints and checks if they are satisfiable.

In general, because of the interaction of length constraints and regular expressions, the length assignment chosen in step 2 might not correspond to satisfiable contents constraints, even when a different length assignment would. So if step 3 fails to find a satisfying assignment, the algorithm returns to step 2 to generate a new length assignment (distinct from any tried previously). Steps 2 and 3 repeat until the solver finds a satisfying assignment, or it has tried all possible length assignments (up to the length bound $k$).

**Step 1: Translating concatenation constraints.**. The intuition behind Kudzu's handling of concatenation constraints is that for a constraint $S_1 = S_2 \circ S_3$, it would be sufficient to ensure that $S_2$ comes immediately before $S_3$ in the final character array, and to lay out $S_1$ as overlapping with $S_2$ and $S_3$ (so that $S_1$ begins at the same character as $S_2$ and ends at the same character as $S_3$). This overlapping layout also has the advantage of reducing the total length of bit-vectors required. Each concatenation constraint suggests an ordering relation among the string variables, but it might not be possible to satisfy all such ordering constraints simultaneously.

To systematically choose an ordering, the solver builds

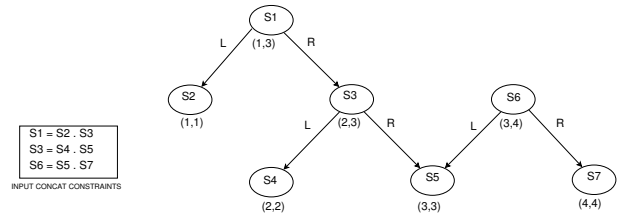


Figure 5: A sample concat graph for a set of concatenation constraints. The relative ordering of the strings in the final character array is shown as start and end positions in parentheses alongside each node.

a graph of concatenation constraints (a *concat graph* for short). The graph has a node for each string variable, and for each constraint $S_1 = S_2 \circ S_3$, $S_2$ and $S_3$ are the left and right children (respectively) of $S_1$. An example of such a graph is shown in Figure 5. Without loss of generality, we can assume that the graph is acyclic: if there is a cycle from $S_1$ to $S_2$ to $S_3$ ... back to $S_1$, then $S_1 = S_2 \circ S_3 \circ \cdots \circ S_1$ (or some other order), so all the variables other than $S_1$ must be the empty string, and can be removed from the constraints. (In our applications the constraints will in any case be acyclic by construction.) Given this graph, the algorithm then chooses the relative ordering of the strings in the character array by assigning start and end positions to each node with a post-order traversal of the graph. (In Figure 5, these positions are shown in parentheses next to each node.)

For the layout generated by the algorithm to be correct, the concat graph must be a DAG in which each internal node has exactly two children, and those children are adjacent in the layout. (This implies that the graph is planar.) The graph may not have these properties at construction; for instance, Figure 6 gives a set of example constraints with contradictory ordering requirements: $S_2$ cannot be simultaneously to the left and to the right of $S_3$. The algorithm resolves such requirements by duplicating a subtree of the graph (for instance as shown in the right half of Figure 6). To maintain the correct semantics, the algorithm adds string equality constraints to ensure that any duplicated strings have the same contents as the originals. The algorithm performs duplications to ensure that the graph satisfies the correctness invariant, but our current algorithm does not attempt to perform the minimal number of copies (for instance, in Figure 6 it would suffice to copy either only $S_2$ or only $S_3$), which in our experience has not hurt the solver's performance.

**Step 2: Finding a satisfiable length assignment.** Each string variable $S$ has an associated length variable $L_S$. Each core string constraint implies a corresponding constraint on the lengths of the strings, as detailed in Table I. For the regular expression containment constraint ($S_1 \in R$), the set of possible lengths is an *ultimately periodic set*:

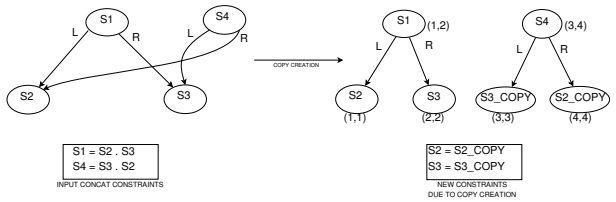


Figure 6: A set of concat constraints with contradictory ordering requirements. Nodes are duplicated to resolve the contradiction.

whether a length is possible depends only on its remainder when divided by a fixed period, except for a finite set of exceptions. (Yu *et al.* use the equivalent concept of a semi-linear set in a conservative automaton-based approach [32].) The details of computing this set are covered in the literature [22]; we note that such sets can be conveniently represented with our SMT solver since it supports a modulus operation. At each iteration of step 2, the solver conjoins the length constraints corresponding to all of the original string constraints, along with a constraint to rule out each length assignment that had previously been tried, and passes this formula to the SMT solver. If it returns a satisfying assignment, it represents a new length assignment to try; if the constraint is unsatisfiable, then so were the original string constraints.

| Core Constraint | Implication on lengths |
|---|---|
| $S_1 = S_2 \circ S_3$ | $L_{S_1} = L_{S_2} + L_{S_3}$ |
| $S_1 \in R$ | $L_{S_1} \in \mathit{LengthSet}(R)$ |
| $S_1 = S_2$ | $L_{S_1} = L_{S_2}$ |
| $\texttt{length}(S_1) \diamond i$ | $L_{S_1} \diamond i$ |
| $\texttt{length}(S_1) \diamond \texttt{length}(S_2)$ | $L_{S_1} \diamond L_{S_2}$ |

Table I: Length constraints implied by core string constraints, where $L_S$ is the length of a string $S$, and $\diamond$ ranges over the operators $\{<, \leq, =, \geq, >\}$.

It is not necessary for correctness that the length abstraction performed by the solver be precise, but determining precise length bounds improves performance by avoiding wasted iterations. In the complete system, the integer constraints over lengths are solved together with integer constraints arising directly from the original program, discussed in Section VI. In our experience it is important for good performance to solve these two sets of integer constraints together. The two sets of constraints may be interrelated, and solving them together prevents the solver from wasting time exploring solutions that satisfy one set but not the other.

**Step 3: Encoding as bitvectors.** Given the array layout and lengths computed in steps 1 and 2, the remaining constraints over the contents of strings can be expressed as constraints over fixed-size bit-vectors. String equality translates directly into bit-vector equality. For the encoding of regular expression constraints, we reuse part of the implementation of the HAMPI tool [17]. At a high-level, the translation first unrolls uses of the Kleene star $*$ in a regular expression into a finite number of repetitions (never more than the string length). Next, where the regular expression has concatenation, HAMPI determines all possible combinations of lengths that sum to the total length, and instantiates each as a conjunction of constraints. Along with the alternations that appeared in the original regular expression, each of these conjunctions also represents an alternative way in which the regular expression could match the string. To complete the translation, the choice between all of these alternatives is represented with a disjunction. (See [17] for a more detailed explanation and some optimizations.)

HAMPI supports only a single, fixed-length input, so we invoke it repeatedly to translate each constraint between a regular expression and a string into an STP formula. We then combine each of these translations with our translations of other string contents constraints (e.g., string equality), and conjoin all of these constraints so that they apply to the same single long character array. It is this single combined formula that we pass to the SMT solver (STP) to find a satisfying assignment.

## VI. Reducing JavaScript to String Constraints

In this section we describe our tool's translation from JavaScript to the language of our constraint solver, focusing on the treatment of string operations. We start by giving the full constraint language the solver supports, then describe our general approach to modeling string operations, our use of concrete values from the dynamic trace, and the process of translating real regular expressions into textbook-style ones.

**Full constraint language.** The core constraint language presented in Section IV captures the essence of our solving approach, but it excludes several features for simplicity, most notably integer constraints. The full constraint language supported by our solver supports values of string, integer, and boolean types, and its grammar is given in Figure 8, along with its type system in Figure 7. The additional constraints are solved at step 2 of the string solution procedure, together with the integer constraints on the lengths of strings. To match common JavaScript implementations (which reserve a bit as a type tag), we model integers as 31-bit signed bit-vectors in our SMT solver, which supports all the integer operations that JavaScript does. The solver replaces each `toString` constraint with the appropriate string once a value for its argument is selected: for instance, if $i$ is given the value 12, $\texttt{toString}(i)$ is replaced with "`12`".

**JavaScript string operations.** JavaScript has a large library of string operations, and we do not aim to support every operation, or the full generality of their behavior. Beyond the engineering challenge of building such a complete translation, having very complex symbolic translations for common operators would likely cause the system to bog down, and

$$
\begin{array}{rcl}
\tau & ::= & \mathit{string} \mid \mathit{int} \mid \mathit{bool} \\
\mathit{ConstRegex} & ::= & \mathit{Regex} \mid \mathit{CapturedBrack}(R, i) \mid \\
 & & \mathit{BetweenCapBrack}(R, i, j)
\end{array}
$$

Figure 7: Type system for the full constraint language

$$
\begin{array}{rcl}
S_1 : \mathit{string} & = & (S_2 : \mathit{string}) \circ (S_3 : \mathit{string}) \circ \cdots \\
I_1 : \mathit{int} & = & \texttt{length}(S : \mathit{string}) \\
S_1 : \mathit{string} & \in & R : \mathit{ConstRegex} \\
S_1 : \mathit{string} & \notin & R : \mathit{ConstRegex} \\
I_1 : \mathit{int} & = & (I_2 : \mathit{int})\ \{+, -, \cdot, /\}\ (I_3 : \mathit{int}) \\
B_1 : \mathit{bool} & = & (A_1 : \tau)\ \{=, \neq\}\ (A_2 : \tau) \\
B_1 : \mathit{bool} & = & (I_1 : \mathit{int})\ \{<, \leq, \geq, >\}\ (I_2 : \mathit{int}) \\
B_1 : \mathit{bool} & = & \neg(B_2 : \mathit{bool}) \\
B_1 : \mathit{bool} & = & (B_1 : \mathit{bool})\ \{\wedge, \vee\}\ (B_2 : \mathit{bool}) \\
S_1 : \mathit{string} & = & \texttt{toString}(I_1 : \mathit{int})
\end{array}
$$

Figure 8: Grammar and types for the full constraint language including operations on strings, integers, and booleans.

```
function validate (input) {
 //input = '{"action":"","val":""}';
 mustMatch = '{]:],]:]}';
 re1 =/\\(?:["\\\/bfnrt]|u[0-9a-fA-F]{4})/g;
 re2 =/"[^"\\\n\r]*"|true|false|null|
      -?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?/g;
 re3 = /(?:^|:|,)(?:\s*\[)+/g;
 rep1 = str.replace(re1, "@");
 rep2 = rep1.replace(re2, "]");
 rep3 = rep2.replace(re3, "");
 if(rep3==mustMatch) { eval(input); return true; }
 return false;              }
```

Figure 9: Example of a regular-expression-based validation check, adapted from a real-world JavaScript application. This illustrates the complexity of real regular expression syntax.

the generality would usually be wasted. Instead, our choice has been to model the string operations that occur commonly in web applications, and the core aspects of their behavior. For other operations and behavior aspects our tool uses values from the original execution trace (described further below), so that they are accurate with respect to the original execution even if the tool cannot reason symbolically about how they might change on modified executions. The detailed translation from several common operators (a subset of those supported by our implementation) to our constraint language is shown in Table II.

**Using dynamic information.** One of the benefits of dynamic symbolic execution is that it provides the flexibility to choose between symbolic values (which introduce generality) and concrete values (which are less general, but guaranteed to be precise) to control the scope of the search process. Our tool's handling of string constraints takes advantage of concrete values from the dynamic traces in several ways. An example is string `replace`, which is often used in sanitization to transform unsafe characters into safe ones. Our translation uses a concrete value for the number of occurrences of the searched-for pattern: if a pattern was replaced six times in the original run, the tool will search for other inputs in which the pattern occurs six times. This sacrifices some generality (for instance, if a certain attack is only possible when the string appears seven times). However, we believe this is a beneficial trade-off since it allows our tool to analyze and find bugs in many uses of `replace`. For comparison, most previous string constraint solvers do not support `replace` at all, and adding a `replace` that applied to any number of occurrences of a string (even limited to single-character strings) would make our core constraint language undecidable in the unbounded case [6].

**Regular expressions in practice.** The "regular expressions" supported by languages like JavaScript have many more features than the typical definition given in a computability textbook (or Figure 2). Figure 9 shows an example (adapted from a real web site) of one of many regular expressions Kudzu must deal with. Kudzu handles a majority of the syntax for regular expressions in JavaScript, which includes support for (possibly negated) character classes, escaped sequences, repetition operators (`{n}` `/?` `/*` `/+`) and submatch extraction using capturing parentheses. Kudzu keeps track of the nesting of capturing parentheses, so that it can express the relation between the input string and the parts of it that match the captured groups (as shown in Table II). Kudzu does not currently support back-references (they are strictly more expressive than true regular expressions), though if we see a need in the future, many uses of back-references could be translated using (non-regular) concatenation constraints.

## VII. Experimental Evaluation

We have built a full-implementation of Kudzu using the WebKit browser, with 650, 7430 and 2200 lines of code in the path constraint extraction component, constraint solver, and GUI explorer component, respectively. The system is written in a mixture of C++, Ruby, and OCaml languages.

We evaluate Kudzu with three objectives. One objective is to determine whether Kudzu is practically effective in exploring the execution space of real-world applications and uncovering new code. The second objective is to determine the effectiveness of Kudzu as a stand-alone vulnerability discovery tool — whether Kudzu can automatically find client-side code injection vulnerabilities and prune away false reports. Finally, we measure the efficiency of the constraint solver. Our evaluation results are promising, showing that Kudzu is a powerful system that finds previously unknown vulnerabilities in real-world applications fully automatically.

### A. Experiment Setup

We select 18 subject applications consisting of popular iGoogle gadgets and AJAX applications, as these were studied by our previous tool FLAX [26]. FLAX assumes availability of an external (manually developed) test suite to seed its testing; in contrast, Kudzu automatically generates a much more comprehensive test suite and finds the points

| JavaScript operation | Reduction to our constraint language |
|---|---|
| $S_1 : \mathit{string} = \text{charAt}(S : \mathit{string}, I : \mathit{int})$ | $(((L_S = \texttt{length}(S)) \wedge (I \geq 0) \wedge (I < L_S)$<br>$\wedge (S_1 = T_1 \circ T_2 \circ T_3) \wedge (T_2 = S)$<br>$\wedge (I = \texttt{length}(T_1) + 1))$<br>$\vee (((I \geq L_S) \vee (I < 0)) \wedge (S_1 = \text{“”})))$ |
| $I_1 : \mathit{int} = \text{charcodeAt}(S : \mathit{string}, I : \mathit{int})$ | $(((L_S = \texttt{length}(S)) \wedge (I \geq 0) \wedge (I < L_S) \wedge (S_1 = T_1 \circ T_2 \circ T_3) \wedge (T_2 = S)$<br>$\wedge (I = \texttt{length}(T_1) + 1) \wedge (S_1 = \texttt{toString}(I_1)))$<br>$\vee (((I \geq L_S) \vee (I < 0)) \wedge (S_1 = \texttt{toString}(I_1)) \wedge (S_1 = \text{“}\texttt{NaN}\text{”})))$ |
| $S : \mathit{string} = \text{concat}(S_1 : \mathit{string}, S_2 : \mathit{string}, \ldots, S_k : \mathit{string})$ | $(S = S_1 \circ S_2 \circ \cdots \circ S_k)$ |
| $I : \mathit{int} = \text{indexOf}(S : \mathit{string}, s : \mathit{string}, \mathit{startpos} : \mathit{int})$ | $(((I \geq 0) \wedge ((\mathit{startpos} < \texttt{length}(S)) \wedge (\mathit{startpos} \geq 0)) \wedge (S = S_1 \circ S_2)$<br>$\wedge (\mathit{startpos} = \texttt{length}(S_1)) \wedge (S_2 = T_1 \circ s \circ T_3) \wedge (I = \texttt{length}(T_1))$<br>$\wedge (T_1 \notin \mathit{Regex}(\texttt{.*}s\texttt{.*})))$<br>$\vee ((I < 0) \wedge ((\mathit{startpos} < \texttt{length}(S)) \wedge (\mathit{startpos} \geq 0)) \wedge (S \notin \mathit{Regex}(\texttt{.*}s\texttt{.*})))$<br>$\vee (\neg((\mathit{startpos} < \texttt{length}(S)) \wedge (\mathit{startpos} \geq 0)) \wedge (S_1 = \texttt{toString}(I_1)) \wedge (S_1 = \text{“}\texttt{NaN}\text{”})))$ |
| $I : \mathit{int} = \text{lastIndexOf}(S : \mathit{string}, s : \mathit{string}, \mathit{startpos} : \mathit{int})$ | $(((I \geq 0) \wedge ((\mathit{startpos} < \texttt{length}(S)) \wedge (\mathit{startpos} \geq 0)) \wedge (S = S_1 \circ S_2) \wedge (\mathit{startpos} = \texttt{length}(S_1))$<br>$\wedge (S_2 = T_1 \circ s \circ T_3) \wedge (I = \texttt{length}(T_1)) \wedge (T_3 \notin \mathit{Regex}(\texttt{.*}s\texttt{.*})))$<br>$\vee ((I < 0) \wedge ((\mathit{startpos} < \texttt{length}(S)) \wedge (\mathit{startpos} \geq 0)) \wedge (S \notin \mathit{Regex}(\texttt{.*}s\texttt{.*})))$<br>$\vee (\neg((\mathit{startpos} < \texttt{length}(S)) \wedge (\mathit{startpos} \geq 0)) \wedge (S_1 = \texttt{toString}(I_1)) \wedge (S_1 = \text{“}\texttt{NaN}\text{”})))$ |
| $[S_1, S_2, \ldots, S_k] : \mathit{string\ list} = \text{match}(S : \mathit{string}, r : \mathit{ConstRegex})$<br>(non-greedy) | $(((k > 0) \wedge (S \in r) \wedge ((S_1 \in \mathit{CapturedBrack}(r, 1)) \vee (S_1 = \text{“”}))$<br>$\wedge (S = T_0 \circ S_1 \circ T_1 \circ \cdots \circ S_k \circ T_k) \wedge (\bigwedge_{i=0}^{k} T_i \in \mathit{BetweenCapBrack}(r, i, i+1)))$<br>$\wedge \cdots \wedge ((S_k \in \mathit{CapturedBrack}(r, k)) \vee (S_k = \text{“”})))$<br>$\vee ((k \leq 0) \wedge (S \notin \mathit{Regex}(\texttt{.*}r\texttt{.*}))))$ |
| $[S_1, S_2, \ldots, S_n] : \mathit{string\ list} = \text{match}(S : \mathit{string}, r : \mathit{ConstRegex}, n : \mathit{int})$<br>(greedy match)<br>$n$ is the concrete number of occurrences of strings matching $r$. | $(((S = T_1 \circ M_1 \circ T_2 \circ M_2 \circ \cdots \circ T_n \circ M_n \circ T_{n+1}) \wedge (T_1, T_2, \ldots, T_{n+1} \notin \mathit{Regex}(\texttt{.*}r\texttt{.*}))$<br>$\wedge (M_1, M_2, \ldots, M_n \in \mathit{Regex}(r)))$ |
| $S_1 : \mathit{string} = \text{replace}(S : \mathit{string}, r : \mathit{ConstRegex}, s : \mathit{string}, n : \mathit{int})$<br><br>$n$ is the concrete number of occurrences of strings matching $r$ in $S$. | $((S = T_1 \circ M_1 \circ T_2 \circ M_2 \circ \cdots \circ T_n \circ M_n \circ T_{n+1}) \wedge (T_1, T_2, \ldots, T_{n+1} \notin \mathit{Regex}(\texttt{.*}r\texttt{.*}))$<br>$\wedge (M_1, M_2, \ldots, M_n \in \mathit{Regex}(r)))) \wedge (S_1 = T_1 \circ s \circ T_2 \circ s \circ \cdots \circ T_{n+1})$ |
| $[S_1, S_2, \ldots, S_k] : \mathit{string\ list} = \text{split}(S : \mathit{string}, s : \mathit{string}, n : \mathit{int})$<br>$n$ is the concrete number of occurrences of strings matching $r$. | $(((S = S_1 \circ s \circ S_2 \circ s \circ \cdots \circ S_n \circ s \circ S_{n+1}) \wedge (S_1, S_2, \ldots, S_{n+1} \notin \mathit{Regex}(\texttt{.*}s\texttt{.*}))))$ |
| $I_1 : \mathit{int} = \text{search}(S : \mathit{string}, r : \mathit{ConstRegex})$ | $(((I_1 < 0) \wedge (S \notin \texttt{.*}r\texttt{.*}))$<br>$\vee ((I_1 \geq 0) \wedge (S = T_1 \circ T_2 \circ T_3) \wedge (I_1 = \texttt{length}(T_1)) \wedge (T_2 \in \mathit{Regex}(r)) \wedge (T_1, T_3 \notin (\texttt{.*}r\texttt{.*}))))$ |
| $S_1 : \mathit{string} = \text{substring}(S : \mathit{string}, \mathit{start} : \mathit{int}, \mathit{end} : \mathit{int})$ | $((\mathit{start} \geq 0) \wedge (\mathit{end} < \texttt{length}(S)) \wedge (\mathit{end} \geq \mathit{start}) \wedge (S = T_1 \circ S_1 \circ T_2) \wedge (\mathit{start} = \texttt{length}(T_1))$<br>$\wedge (I_1 = \mathit{end} - \mathit{start}) \wedge (I_1 = \texttt{length}(S_1))$<br>$\vee ((\mathit{start} \geq 0) \wedge (\mathit{end} \geq \texttt{length}(S)) \wedge (\mathit{end} \geq \mathit{start}) \wedge (S = T_1 \circ S_1 \circ T_2) \wedge (\mathit{start} = \texttt{length}(T_1))$<br>$\wedge (L_S = \texttt{length}(S)) \wedge (I_1 = L_S - \mathit{start}) \wedge (I_1 = \texttt{length}(S_1))$<br>$\vee ((\mathit{start} < 0) \wedge (\mathit{end} < \texttt{length}(S)) \wedge (\mathit{end} \geq \mathit{start}) \wedge (S = T_1 \circ S_1 \circ T_2) \wedge (0 = \texttt{length}(T_1))$<br>$\wedge (I_1 = \mathit{end} - \mathit{start}) \wedge (I_1 = \texttt{length}(S_1))$<br>$\vee ((\mathit{start} < 0) \wedge (\mathit{end} \geq \texttt{length}(S)) \wedge (\mathit{end} \geq \mathit{start}) \wedge (S = T_1 \circ S_1 \circ T_2) \wedge (0 = \texttt{length}(T_1))$<br>$\wedge (L_S = \texttt{length}(S))(L_S = \texttt{length}(S_1))$ |
| $B_1 : \mathit{bool} = \text{match}(S, r)$ | $((B_1) \wedge (S \in \mathit{Regex}(r)) \vee (\neg(B_1) \wedge (S \notin \mathit{Regex}(r))))$ |
| $I : \mathit{int} = \text{parseInt}(S)$ | $(S = \texttt{toString}(I)) \wedge ((S = \text{“}\texttt{NaN}\text{”}) \vee (S = \mathit{Regex}(\texttt{[0-9]+})))$ |

Table II: Our reduction from common JavaScript operations to our full constraint language. Capitalized variables may be concrete or symbolic, while lowercase variables (including regular expressions) must take a concrete value. Our implementation supports a larger set of operations, whose translations are similar.

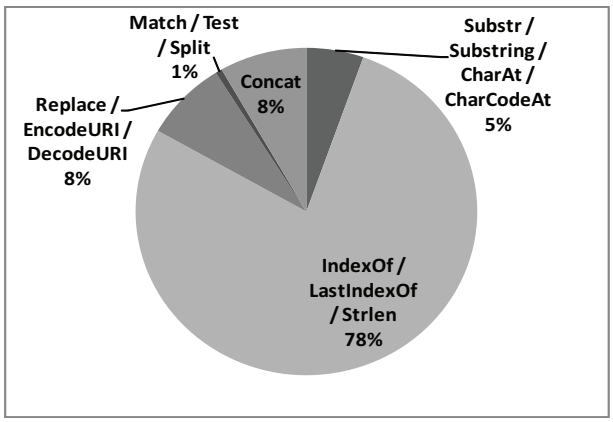


Figure 10: Distribution of string operations in our subject applications.

| Application | # of new inputs | Initial / Final Code Coverage | Bug found |
|---|---|---|---|
| Academia | 20 | 30.27 / 76.47% | ✓ |
| AJAXIm | 15 | 49.58 / 77.67% | ✓ |
| FaceBook Chat | 54 | 26.85 / 76.84% | - |
| ParseUri | 13 | 53.90 / 86.10% | ✓ |
| Plaxo | 31 | 5.72 / 76.43% | ✓ |
| AskAWord | 10 | 29.30 / 67.95 % | ✓ |
| Birthday Reminder | 27 | 59.47 / 73.94% | - |
| Block Notes | 457 | 65.06 / 71.50 % | ✓ |
| Calorie Watcher | 16 | 64.54 / 73.53% | - |
| Expenses Manager | 133 | 61.09 / 76.56% | - |
| Listy | 19 | 65.31 / 79.73% | ✓ |
| Notes LP | 25 | 46.62 / 76.67% | - |
| Progress Bar | 12 | 63.60 / 75.09% | - |
| Simple Calculator | 1 | 46.96 / 80.52% | ✓ |
| Todo List | 15 | 72.51 / 86.41% | ✓ |
| TVGuide | 6 | 30.39 / 75.13% | ✓ |
| Word Monkey | 20 | 14.84 / 75.36% | ✓ |
| Zip Code Gas | 11 | 59.05 / 74.28% | - |
| Average | **51** | **46.95 / 76.68%** | **11** |

Table III: The top 5 applications are AJAX applications, while the rest are Google/IG gadget applications. Column 2 reports the number of distinct new inputs generated, and column 3 reports the increase in code coverage from the initial run to and the final run.

of vulnerability without requiring any external test harness a priori. Further, in our experiments Kudzu discovers 2 new vulnerabilities within a few hours of testing which were missed by the FLAX because of its lack of coverage. In addition, as we show later in this section, many of the generated constraints are highly complex and not suitable for manual inspection or fuzzing, whereas Kudzu either asserts the safety of the validation checks or finds exploits for vulnerabilities in one iteration as opposed to many rounds of random testing.

To test each subject application, we seed the system with the URL of the application. For the gadgets, the URLs are the same as those used by iGoogle page to embed the gadget. We configure Kudzu to give a pre-prepared username and login password for applications that required authentication. We report the results for running each application under Kudzu, capping the testing time to a maximum of 6 hours for each application. All tests ran on a Ubuntu 9.10 Linux workstation with 2.2 GHz Intel dual-core processors and 2 GB of RAM.

### *B. Results*

Table III presents the final results of testing the subject applications. The summary of our evaluation highlights three features of Kudzu: (a) it automatically discovers new program paths in real applications, significantly enhancing code coverage; (b) it finds 2 client-side code injection in the wild and several in applications that were known to contain vulnerabilities; and (c) Kudzu significantly prunes away false positives, successfully discarding cases that do employ sufficient validation checks.

**Characteristics of string operations in our applications.** Constraints arising from our applications have an average of 63 JavaScript string operations, while the remaining are boolean, logical and arithmetic constraints. Figure 10 groups the observed string operations by similarity. The largest fraction are operations like `indexOf` that take string inputs and return an integer, which motivate the need for a solver that reasons about integers and strings simultaneously. A significant fraction of the operations, including `subtring`, `split` and `replace`, implicitly give rise to new strings from the original one, thereby giving rise to constraints involving multiple string variables. Of the `match`, `split` and `replace` operations, 31% are regular expression based. Over 33% of the regular expressions have one or more capturing parentheses. Capturing parentheses in regular expression based match operations lead to constraints involving multiple string variables, similar to operations such as `split`.

These characteristics show that a significant fraction of the string constraints arising in our target applications require a solver that can reason about multiple string variables. We empirically see examples of complex regular expressions as well as concatenation operations, which stresses the need for our solver that handles both word equations and regular expression constraints. Prior to this work, off-the-shelf solvers did not support word equations and regular expressions simultaneously.

**Vulnerability Discovery.** Kudzu is able to find client-side code injection vulnerabilities in 11 of the applications tested. 2 of these were not known prior to these experiments and were missed by FLAX. One of them is on a social-networking application (`http://plaxo.com`) that was missed by our FLAX tool because the vulnerability exists on a page linked several clicks away from the initial post-

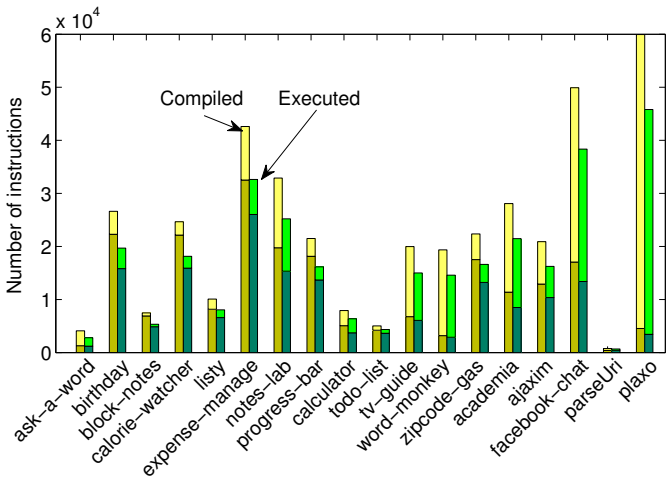


Figure 11: Kudzu code coverage improvements over the testing period. For each experiment, the right bar shows the increase in the executed code from the initial run to total code executed. The left bar shows the increase in the code compiled from initial run to the total code compiled in the entire test period.

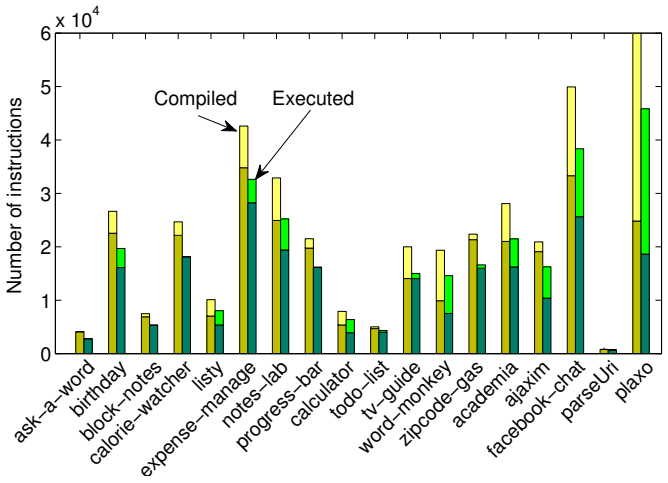


Figure 12: Benefits from symbolic execution alone (dark bars) vs. complete Kudzu (light bars). For each experiment, the right bar shows the increase in the total executed code when the event-space exploration is also turned on. The left bar shows the observed increase in the code compiled when the event-space exploration is turned on.

authentication page. The vulnerable code is executed only as part of a feature in which a user sets focus on a text box and uses it to update his or her profile. This is one of the many different ways to update the profile that the application provides. Kudzu found that only one of these ways resulted in a client-side code injection vulnerability, while the rest were safe. In this particular functionality, the application fails to properly validate a string from a `postMessage` event before using it in an `eval` operation. The application implicitly expects to receive this message from a window hosted at a sub-domain of `facebook.com`; however, Kudzu automatically determines that *any* web principal could inject any data string matching the format `FB_msg:.*{.*}`. This subsequently results in code injection because the vulnerable application fails to validate the origin of the sender and the structure of JSON string before its use in `eval`.

The second new vulnerability was found in a ToDo Google/IG gadget. Similar to the previous case, the vulnerability becomes reachable only when a specific value is selected from a dropdown box. This interaction is among many that the gadget provides and we believe that Kudzu's automatic exploration is the key to discovering this use case. In several other cases, such as AjaxIM, the vulnerable code is executed only after several events are executed after initial sign-in page—Kudzu automatically reaches them during its exploration.

Kudzu did not find vulnerabilities in only one case that FLAX reported a bug. This is because the vulnerability was patched in the time period between our experimental evaluation of FLAX and Kudzu.

**Code and Event-space Coverage.** Table III shows the code coverage by executing the initial URL, and the final coverage after the test period. Measuring code coverage in a dynamically compiled language is challenging because all the application code is not known prior to the experiments. In our experiments, we measured the total code compiled during our experiments and the total code executed [3].

Table III shows an average improvement of over 29% in code coverage. The coverage varies significantly depending on the application. Figure 11 provides more detail. On several large applications, such as Facebook Chat, AjaxIM, and Plaxo, Kudzu discovers a lot of new code during testing. Kudzu is able to concretely execute several code paths, as shown by the increase in the right-side bars in Figure 11. On the other less complex gadget applications, most of the relevant code is observed during compilation in the initial run itself, leaving a relatively smaller amount of new code for Kudzu to discover. We also manually analyzed the source code of these applications and found that a large fraction of their code branches were not dependent on data we treat as untrusted.

To measure the benefits of symbolic execution alone, we repeated the experiments with the event-space exploration turned off during the test period and report the comparison to full-featured Kudzu in Figure 12. We consistently observe

[3]One unit of code in our experiments is a JavaScript bytecode compiled by the interpreter. To avoid counting the same bytecode across several runs, we adopted a conservative counting scheme. We assigned a unique identifier to each bytecode based on the source file name, source line number, line offset and a hash of the code block (typically one function body) compiled.

| Application | # of initial events fired | # of total events fired | Total events discovered |
|---|---|---|---|
| Academia | 20 | 78 | 310 |
| AJAXIm | 72 | 481 | 988 |
| FaceBook Chat | 15 | 989 | 1354 |
| ParseUri | 5 | 16 | 17 |
| Plaxo | 88 | 381 | 688 |
| AskAWord | 2 | 8 | 11 |
| Birthday Reminder | 12 | 20 | 20 |
| Block Notes | 7 | 85 | 319 |
| Calorie Watcher | 14 | 18 | 22 |
| Expenses Manager | 10 | 107 | 1473 |
| Listy | 15 | 470 | 638 |
| Notes LP | 10 | 592 | 1034 |
| Progress Bar | 8 | 24 | 36 |
| Simple Calculator | 17 | 34 | 67 |
| Todo List | 8 | 26 | 61 |
| TVGuide | 17 | 946 | 1517 |
| Word Monkey | 3 | 10 | 22 |
| Zip Code Gas | 12 | 12 | 12 |
| Average | **18.61** | **238.72** | **477.17** |

Table IV: Event space Coverage: Column 2 and 3 show the number of events fired in the first run and in total. The last column shows the total events discovered during the testing.

that symbolic execution alone discovers and executes a significant fraction of the application by itself. The event-exploration combined with symbolic execution does perform strictly better than symbolic execution in all but 3 cases. In a majority of the cases, turning on the event-space exploration significantly complements symbolic execution, especially for the AJAX applications which have a significant GUI component. In the 3 cases where improvements are not significant, we found that the event exploration generally either led to off-site navigations or the code executed could be explored by symbolic execution alone. For example, in the parseUri case, same code is executed by text-box input as well as by clicking a button on the GUI.

Table IV shows the increase in number of events executed by Kudzu from the initial run to the total at the end of test period. These events include all keyboard and mouse events which result in execution of event handlers, navigation, form submissions and so on. We find that new events are dynamically generated during one particular execution as well as when new code is discovered. As a result, Kudzu gradually discovers new events and was able to execute approximately 50% of the events it observes during the period of testing.

**Effectiveness.** Kudzu automatically generates a test suite of over 50 new distinct inputs on average for an application in the test period (shown in column 2 of table III).

In the exploitable cases we observed, Kudzu was able to show the existence of a vulnerability with an attack string once it reached the point of vulnerability. That is, its constraint solver correctly determines the sufficiency or insufficiency of validation checks in a single query without manual intervention or undirected iteration. This eliminates false positives significantly in practice. For instance, Kudzu found that the Facebook web application has several uses of `postMessage` data in `eval` constructs, but all uses were correctly preceded by checks that assert that the origin of the message is a domain ending in `.facebook.com`. In contrast, the vulnerability in Plaxo fails to check this and Kudzu identifies the vulnerability the first time it reaches that point. Some of the validation checks Kudzu deals with are quite complex — Figure 9 shows an example which is simplified from a real application. These examples are illustrative of the need for automated reasoning tools, because checking the sufficiency of such validation checks would be onerous by hand and impractical by random fuzzing. Lastly, we point out that like most other vulnerability discovery tools, Kudzu can have false negatives because it may fail to cover code, or because of overly strict attack grammars.

**Constraint Solver Evaluation.** Figure 13 shows the running times for solving queries of various input constraint sizes. Each constraint is a either a JavaScript string, arithmetic, logical, or boolean operation. The sizes of the equations varied from 1 to up to 250 constraints. The solver decides satisfiability of the constraints typically under a second for satisfiable cases. As expected, to assert unsatisfiability, the solver often takes time varying from nearly a second to 50 seconds. The variation is large because in many cases the solver asserts unsatisfiable by asserting the unsatisfiability of length constraints, which is inexpensive because the step of bit-vector encoding is avoided. In other cases, the unsatisfiability results only when the solver determines the unsatisfiability of bit-vector solutions.

Our solver requires only an upper bound on the lengths of input variables, and is able to infer satisfiable lengths of variables internally. In these experiments, we increase the upper bound of the input variables from 10 to 100 characters in steps of 20 each. If the solver asserts unsatisfiability up to an upper bound length of 100, we treat the constraints as unsatisfiable.

## VIII. Related Work

Kudzu is the first application of dynamic symbolic execution to client-side JavaScript. Here, we discuss some related projects that have applied similar techniques to server-side web applications, or have used different techniques to search for JavaScript bugs. Finally, we summarize why we needed to build a new string constraint solver.

**Server-side analysis.** JavaScript application code is similar in some ways to server-side code (especially PHP); for instance, both tend to make heavy use of string operations. Several previous tools have demonstrated the use of symbolic execution for finding SQL injection and reflected or stored cross-site scripting attacks to code written in PHP and Java [1], [18], [30]. However, JavaScript code usually

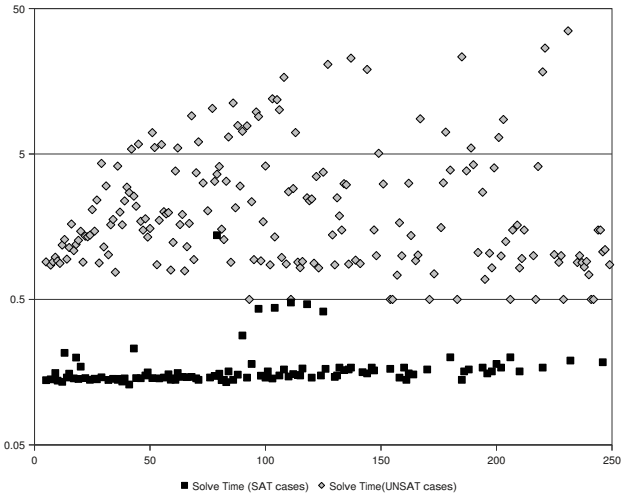


Figure 13: The constraint solver's running time (in seconds) as a function of the size of the input constraints (in terms of the number of symbolic JavaScript operations)

parses its own input, so JavaScript symbolic execution requires more expressive constraints, specifically to relate different strings that were previously part of a single string. Additional challenges unique to JavaScript arise because JavaScript programs take many different kinds of input, some of which come via user interface events.

Like Kudzu, the Saner [2] tool for PHP aims to check whether sanitization routines are sufficient, not just that they are present. However their techniques are quite different: they select paths and model transformations statically, then perform testing to verify some vulnerabilities. Their definition of sanitization covers only string transformations, not validation checks involving branches, which occur frequently in our applications.

**Analysis frameworks for JavaScript.** Several works have recently applied static analysis to detect bugs in JavaScript applications (e.g., [9], [12]). Static analysis is complementary to symbolic execution: if a static analysis is sound, an absence of bug reports implies the absence of bugs, but static analysis warnings may not be enough to let a developer reproduce a failure, and in fact may be false positives.

FLAX uses taint-enhanced blackbox fuzzing to detect if the JavaScript application employs sufficient validation or not [26]; like Kudzu, it searches for inputs to trigger a failure. However, FLAX requires an external test suite to be able to reach the vulnerable code, whereas Kudzu generates a high-coverage test suite automatically. Also, FLAX performs only black-box fuzz testing to find vulnerabilities, while Kudzu's use of a constraint solver allows it to reason about possible vulnerabilities based on the analyzed code.

Crawljax is a recently developed tool for event-space exploration of AJAX applications [23]. Specifically, Crawljax builds a static representation of a Web 2.0 application by clicking elements on the page and building a state graph from the resulting transitions. Kudzu's value space exploration complements such GUI exploration techniques and enables a more complete analysis of the application using combined symbolic execution and GUI exploration.

**String constraint solvers.** String constraint solvers have recently seen significant development, and practical tools are beginning to become available, but as detailed in Section IV-C, no previous solvers would be sufficient for JavaScript, since they lack support for regular expressions [3], [4], [7], string equality [14], or multiple variables [17], which are needed in combination to reason about JavaScript input parsing. In concurrent work, Veanes *et al.* give an approach based on automata and quantified axioms to reduce regular expressions to the Z3 decision procedure [28]. Combined with [4], this would provide similar expressiveness to Kudzu.

## IX. Conclusion

With the rapid growth of AJAX applications, JavaScript code is becoming increasingly complex. In this regard, security vulnerabilities (such as client-side code injection) and analysis of JavaScript code is an important area of research. In this paper, we present the design and development of the first complete symbolic execution based system for exploring the execution space of JavaScript programs. In making the system practical we addressed challenges ranging from designing a more expressive language for string constraints to implementing exploration and replay of GUI events in a web browser. We have implemented our ideas in a tool called Kudzu. Given a URL for a web application, Kudzu automatically generates a high-coverage test suite. We have also applied Kudzu to find client-side code injection vulnerabilities. In a collection of live web applications, it finds 11 vulnerabilities (2 of which were previously unknown) without producing false positives.

## X. Acknowledgments

We thank David Wagner, Adam Barth, Domagoj Babic, Adrian Mettler, Juan Caballero and Pongsin Poosankam for helpful feedback on the paper at various stages. We are also thankful to our anonymous reviewers for suggesting improvements to our work. This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, by the Air Force Office of Scientific Research under Grant No. 22178970-4170, and by the Army Research Office under Grant No. DAAD19-02-1-0389. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Air Force Office of Scientific Research, or the Army Research Office.

## References


[1] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA 2008, International Symposium on Software Testing and Analysis*, 2008.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[3] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers *or* how to stop papers from reviewing themselves. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[4] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[5] H. Bojinov, E. Bursztein, and D. Boneh. XCS: Cross channel scripting and its impact on web applications. In *CCS*, 2009.

[6] J. R. Büchi and S. Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. *Mathematical Logic Quarterly*, 34(4):337–342, 1988.

[7] J. Caballero, S. McCamant, A. Barth, and D. Song. Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries. Technical Report UCB/EECS-2009-36, EECS Department, University of California, Berkeley, March 2009.

[8] A. Chandra, J. Halpern, A. Meyer, and R. Parikh. Equations between regular terms and an application to process logic. In *Proceedings of the 13th annual ACM Symposium on Theory of Computing (STOC)*, pages 384–390, May 1981.

[9] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *PLDI*, 2009.

[10] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification, 19th International Conference (CAV)*, pages 519–531, July 2007.

[11] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security*, Feb. 2008.

[12] S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code. In *Usenix Security*, 2009.

[13] R. Hansen. XSS cheat sheet. http://ha.ckers.org/xss.html.

[14] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 188–198, June 2009.

[15] H. B. Hunt III. The equivalence problem for regular expressions with intersection is not polynomial in tape. Technical Report 73-161, Cornell University Department of Computer Science, Mar. 1973.

[16] J. Karhumäki, W. Plandowski, and F. Mignosi. The expressibility of languages and relations by word equations. In *Automata, Languages and Programming, 24th International Colloquium, (ICALP)*, pages 98–109, 1997.

[17] A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[18] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *30th International Conference on Software Engineering (ICSE)*, May 2009.

[19] A. Klein. DOM based cross site scripting or XSS of the third kind. Technical report, Web Application Security Consortium, 2005.

[20] M. Martin and M. S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *17th USENIX Security Symposium*, 2008.

[21] Y. Matiyasevich. Word equations, Fibonacci numbers, and Hilbert's tenth problem (extended abstract). In *Workshop on Fibonacci Words*, Turku, Finland, Sept. 2006.

[22] A. B. Matos. Periodic sets of integers. *Theoretical Computer Science*, 127(2):287–312, May 1994.

[23] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling ajax by inferring user interface state changes. In *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*, 2008.

[24] W. Plandowski. Satisfiability of word equations with constants is in PSPACE. In *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 495–500, Oct. 1999.

[25] Samy. I'm popular. Description of the MySpace worm by the author, including a technical explanation., Oct 2005.

[26] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *17th Annual Network & Distributed System Security Symposium, (NDSS)*, 2010.

[27] TwitPwn. DOM based XSS in Twitterfall. 2009.

[28] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2010.

[29] W3C. HTML 5 specification. http://www.w3.org/TR/html5/.

[30] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, 2008.

[31] XML path language 2.0. http://www.w3.org/TR/xpath20/.

[32] F. Yu, T. Bultan, and O. H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 322–336, Mar. 2009.


## Appendix

### A. Expressiveness

Each of the following constraint types allows new sets of strings to be defined:

- The theory of just concatenation constraints $v_1 = v_2 \circ v_3$ (plus constant strings) is equivalent to the theory of word equations, which is known to be neither a subset nor a superset of the theory of regular expressions in terms of expressiveness [16]. For instance, the set of strings that consist of a single string repeated $\{ww : w \in \Sigma^*\}$ can be easily expressed with a word equation, but is not regular.
- Conversely, the regular languages expressed by regular expressions include sets of strings that cannot be expressed solely with word equations. An example given

in [16] is the set of strings consisting of a and b over a three-letter alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, which is represented by the regular expression $(\mathtt{a}|\mathtt{b})^\star$.
- The constraint $\mathtt{length}(v_1) = \mathtt{length}(v_2)$ also adds expressiveness, since the language $\{u\$v : |u| = |v|\}$ is not regular, and the relation of two strings having equal length also can not be expressed in word equations for any non-unary alphabet [16].

### B. Complexity

The problem of deciding Kudzu's core constraint language is at least PSPACE-hard, which can be seen by adapting the construction given by Hunt [15] to prove the same hardness for the emptiness problem for regular expressions with an intersection operator. The construction is a reduction from the acceptance problem for a linearly-bounded automaton; since it uses the intersection operator only at the top level of the constructed expression, it can also be expressed as a conjunction of regular expression constraints in our language. (The best known algorithms for the word equation problem are also in PSPACE [24], but the best known lower bound for that fragment is only NP-hardness.)

On the upper bound side, the complexity of our deciding our constraints is still an open problem. The fragment consisting of word equations and length equalities is known to be strictly more expressive that word equations alone [6], [16], but its decidability has been mentioned as open by several authors [4], [6], [21]. Our core constraint language does not contain any features that obviously cause it to be undecidable, but it is strictly more expressive than fragment of word equations and length equalities, so its decidability is also an open problem.

The potential difficulty of deciding the unbounded version of our core constraint language (even PSPACE problems are usually infeasible at scale) motivates our decision in the practical system to search just for solutions using strings of lengths bounded by a user-specified parameter. The bounded version of the constraint decision problem is NP-complete (where the size of the input is measured by the bound, formally by specifying the bound in unary). The reduction from SAT is immediate, since the constraint language includes arbitrary boolean combinations, and the bounded language is in NP because solutions (of size proportional to the bound) can be checked in polynomial time using standard regular expression matching algorithms.

### C. Correctness

We discuss the correctness of our algorithm in terms of soundness, completeness (up to the bound), and termination. These properties build on the corresponding properties for the underlying bit-vector solver and the translation from regular expressions to bit-vectors, which have already been established.

**Concatenation constraints.** The key invariant established by the translation from concatenation constraints to an adjacency ordering is that for each original constraint $S_1 = S_2 \circ S_3$, the first $\mathtt{length}(S_2)$ characters of $S_1$ should be constrained to be equal to the characters of $S_2$, and similarly between the remaining $\mathtt{length}(S_3)$ characters of $S_1$ and $S_3$. When the nodes are adjacent in the concatenation graph, this holds because the characters are in fact the same characters; when nodes are copied to resolve ordering conflicts, the invariant holds because the copies are constrained to be equal to the originals.

**Length abstraction.** The key property of the length constraints synthesized in step 2 is that they be a sound abstraction of the possible lengths of strings that could satisfy the constraints: for any strings that satisfy a string constraint, their lengths must satisfy the corresponding length constraint. It is not necessary for correctness that the abstraction be precise (that every length correspond to a possible string solution), though precision improves the solver's performance by pruning the search space of lengths. For the concatenation, string equality, and length-related constraints, the correctness of the length abstractions is immediate. For a detailed discussion of the operation *LengthSet*$(R)$ we refer readers to [22].

**Soundness.** For soundness, suppose that the algorithm produces a solution that assigns string $s_i$ with length $l_i$ to each string variable $S_i$ that appears in a constraint set. By the soundness of the bit-vector solver, the solution corresponds to an assignment of truth values to each string constraint in the formula that gives the formula the boolean result *true*. Let the *constraint literals* be a set which contains $C$ for each constraint $C$ assigned true, and $\neg C$ for each constraint $C$ assigned false. By the soundness of the bit-vector solver, the lengths $l_i$ are a correct solution to the length constraints, so the length-related constraint literals will be satisfied. By the soundness of the translation of regular expression constraints and the bit-vector solver, $s_i \in L(R_i)$ for each constraint $S_i \in R_i$, so each regular expression constraint literal is satisfied. Finally, for concatenation and string equality constraints, step 1 guaranteed that the corresponding characters were either identified or constrained to be equal, so by construction or by the soundness of the bit-vector solver (respectively), their constraint literals will be satisfied. Thus the assignment produced by the algorithm gives an appropriate value to each constraint literal such that the constraint formula is satisfied.

**Bounded completeness.** For bounded completeness, suppose to the contrary that there exists a set of solution strings $s_i$ with lengths $l_i$ for a set of constraints, but that the algorithm when run on those constraints prints `Unsatisfiable`. Consider two cases, according to whether the lengths $l_i$ were ever returned by the procedure *SolveLengths* during execution. If the lengths were returned

on some iteration $j$, then by the bounded completeness of the regular expression translation, and the correctness of the translation of string equality and concatenation constraints, the bit-vector constraints for those lengths must be satisfiable by the string values $s_i$. However, the algorithm could only have printed `Unsatisfiable` if the bit-vector solver found the bit-vector constraints unsatisfiable, contradicting its completeness. On the other hand, suppose that the lengths $l_i$ were never returned by *SolveLengths*. Because the program printed `Unsatisfiable`, the final call to *SolveLengths* must have returned *UNSAT* when supplied with a set of length constraints $L$, and some number of failed length assignments not including the length assignment $l_i$. By the soundness of the length abstraction, the lengths $l_i$ satisfy $L$, but since the $l_i$ assignment satisfies $L$ and was not excluded, the completeness of the bit-vector solver implies that it should have been returned by *SolveLengths*, contradicting our assumption that it was never returned by *SolveLengths*. Thus neither case is possible, so we reject the assumption and conclude that if a solution exists, the algorithm will not print `Unsatisfiable`.

**Termination.** For termination, only the main **while** loop must be considered, since all of the subroutines are guaranteed to terminate. To see that the loop terminates, we observe that the set *FailLenDB* of failed length assignments grows on each iteration of the main loop, since when *SolveLengths* gives *SAT*, it always returns a new length assignment distinct from those previously in *FailLenDB*. On the other hand, the length assignments returned by *SolveLengths* always satisfy the length bounds in $C$. But because the length of each string is a non-negative integer no greater than the corresponding length bound, there are only finitely many possible length assignments that satisfy the bounds. Thus the correctness of *SolveLengths* and the fact that *FailLenDB* grows monotonically together guarantee that *SolveLengths* will eventually return *UNSAT*, causing the algorithm to terminate.