Propagation Effects in Event-Driven JavaScript

Aaron Bembenek  
bembenek@g.harvard.edu

Todd Lubin  
tlubin@college.harvard.edu

Lily Tsai  
liliantsai@college.harvard.edu

I. SOFTWARE RELIABILITY ISSUE

Real-world, web-based JavaScript programs tend to be heavily event-driven with many callback functions that are invoked by user interaction. Because these callback functions modify the same state but can be called in a largely arbitrary order, it is difficult for developers to reason about how changes to the code of one callback function will effect the functionality of other callback functions in the same script. Nonetheless, there are currently no tools available to effectively and efficiently evaluate patches of this nature. We propose ???, a tool that uses a form of compositional/targeted symbolic execution to analyze changes introduced through patching JavaScript callback functions.

There is a substantial corpus of work on using symbolic execution to evaluate the effect of patches, both in terms of whether a patch actually fixes the targeted problem and whether a patch has unintended consequences that manifest elsewhere in the code (a common phenomenon in practice). However, the previously proposed techniques largely focus on code written in static languages, such as Java and C/C++. Only recently have symbolic execution engines been developed for code written in more dynamic languages like JavaScript. (JavaScript has the additional challenge of being heavily string-based, which can be a source of trouble for constraint solvers.) Jalangi is a dynamic analysis platform for JavaScript code that is packaged with a concolic execution engine, but it is not instrumented to handle the event-driven nature of real-world JavaScript applications. Kudzu uses symbolic string constraints to evaluate code injection vulnerabilities in scripts and has limited support for event-driven code. SymJS fully embraces the nature of event-driven code in its symbolic execution engine, but it is focused on achieving high code coverage and does not evaluate the effect of modifications to a script.

II. IDEAL GOAL

The ideal goal of the project is to develop a tool that can report the propagation effect of small patches in a single JavaScript callback function. This tool will take in an HTML file and two JavaScript files (where one has a small change in a callback function) and find sequences of callbacks such that running the sequence in the original file and the patched file have propagation effects. Let us formally specify propagation effects. A JavaScript file is thought of as a collection of callback functions. A run of a given file is a finite sequence of executions of callback functions within a context (complete state of DOM tree and global variables). For example, a run may be the user typing “CS260” and then clicking a button with the DOM tree and global variables all taking their initial values. A run deterministically takes a context to a new context. Most likely there exists a context such that running the original function, $f$, and patched function, $f'$, in this context will result in two differing contexts (otherwise the code change didn’t do anything interesting). These different contexts have a set of atomic differences $\Delta = \{\delta_1, ..., \delta_n\}$. A propagation effect is some $\delta \notin \Delta$ such that $\delta$ is an atomic difference on output contexts of two runs, one of which calls $f$ and not $f'$ while the other calls $f'$ and not $f$.

Let us be more concrete with our approach. A user makes a change to a callback function $f$ (e.g. adds a line of code), creating a new function $f'$. Our tool works in 4 phases:

1) Constraint generation: Generate a set of context constraints that must be met in order for the patched code to be executed.

2) Find satisfying context: Find a feasible context $C_s$ that satisfies the generated set of constraints. If the initial context $C_0$ (the initial state of the DOM tree and global variables) satisfies all the constraints, then this phase is trivial. Otherwise, we “transform” $C_0$ into $C_s$ by analyzing the effect of certain callbacks on transforming a context. This is a sort of record and replay approach where we get our hands on a $C_s$ by actually executing callbacks and observing how the context changes. In general we acknowledge this could be a very challenging aspect of the project that require a well-tuned set of heuristics.

3) Execute patched code: Execute $f$ and $f'$ in the context $C_s$. This will result in two contexts $C_f$ and $C_{f'}$ that have some set $\Delta$ of atomic differences.

4) Propagation Analysis: Find a sequence of executions of callback functions such that the run takes takes $C_f$ to $C_p$ and $C'_{f'}$ to $C'_{p'}$ where $C_p$ and $C'_{p'}$ have some atomic change $\delta$ that is not in the set of atomic changes of $C_f$ and $C'_{f'}$. This is also a very challenging aspect of the project as it can almost be thought of as symbolic execution where the input is an interleaving of function calls. SymJS does a very similar thing to generate test cases that achieve high code coverage. We look to employ similar heuristics and analyzes to guide the search. For example, if a button is hidden, there is no reason to execute the on-click callback for that button. This search is entirely dynamic and involves holding onto two differing contexts and running callbacks in
We have two goals for our one week project:

- **Constraint generation:** We plan to be able to take a function along with the patched function and symbolically generate a set of constraints on a context that are necessary in order for the patched code to actually be executed. This can be accomplished with a forward or backward approach, as discussed in the following **Technologies** section. The initial context then will be “transformed” into a context that satisfies the set of generated constraints. In simple cases, there may exist no constraints as the patched code will always be executed (which is why we ignore it for the one week project).

- **Record and Replay Engine:** We plan to build the record and replay infrastructure. This involves building the framework that allows for the execution of callback functions in a script within an arbitrary context. As discussed in the **Technologies** section, we will try to leverage Jalangi for this so we don’t have to entirely reinvent the wheel.

### V. Worries

Our worries for this project include concern that our tool might generate a lot of uninteresting results, or results that are obvious to the programmer who introduced the patch. On the other hand, it is also possible for our tool to report no propagation effects when they actually exist, since we may be working with a constraint-satisfying context that does not exhibit propagation effects even while there exists another constraint-satisfying context that does. In addition, we are unsure how to evaluate our project because we might have to rely on our own invented examples to determine if our tool works. One idea we have is to find some open source JavaScript applications and find diffs in commits, using these diffs as the patches to analyze. Our tool also faces a huge problem with path explosion, and we hope to find some heuristics and take ideas from related papers to constrain the number of paths we will explore. In addition, we recognize that most real-world web development commonly utilizes external libraries like jQuery. In order to make this tool useful, it should work seamlessly with libraries like jQuery. Because we are doing concolic execution, this actually shouldn’t be too bad to just execute the JQuery commands dynamically as they appear.

### VI. Technologies

There are a lot of technologies we are going to try to leverage (or at least get inspiration from) to build ??. Primarily, Jalangi has a record and replay engine for JavaScript code. We can use or take inspiration from this as we build our own. In order to find the constraints on a context needed to trigger execution of the patched code we can look at call-chain-backward- symbolic execution (CCBSE). Alternatively, we can execute the function symbolically to explore paths and find the corresponding constraints when we hit the target patched code. These two approaches correspond to working forwards from the beginning of the function versus going backwards.
from the changed code in order to find the constraints. In considering interleaving of functions, we can learn something from SymJS which implements a series of dynamic heuristics in order to prevent path explosion. Unfortunately, SymJS is not openly available online, so we are left with only their ideas and descriptions within their research paper.

VII. PAPERS TO READ

• Directed Symbolic Execution
• Shadow Symbolic Execution
• Differential Symbolic Execution
• Partition-Based Symbolic Execution
• Demand-Driven Compositional Symbolic Execution
• SymJS: symbolic execution of an event-driven JavaScript program

VIII. COLLABORATION PLAN

We will work together for at least the one-week project to build the foundation of our project, and then divide the work afterward.