2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

Analysis of JavaScript Web Applications Using SAFE 2.0

Jihyeok Park
KAIST
jhpark0223@kaist.ac.kr

Yeonhee Ryou
KAIST
ryou770@kaist.ac.kr

Joonyoung Park
KAIST
gmb55@kaist.ac.kr

Sukyoung Ryu
KAIST
sryu.cs@kaist.ac.kr

***Abstract*—JavaScript has been the language for web applications, and the growing prevalence of web environments in various devices makes JavaScript web applications even more ubiquitous. However, because JavaScript and web environments are extremely dynamic, JavaScript web applications are often vulnerable to type-related errors and security attacks. To lessen the problem, researchers have developed various analysis techniques in different analyzers, but such analyzers are not especially aimed for ease of use by analysis developers. In this paper, we present SAFE 2.0, a scalable analysis framework for ECMAScript especially designed as a playground for advanced research in JavaScript web applications. SAFE 2.0 is light-weight, which supports pluggability, extensibility, and debuggability.**

Demo video: `https://youtu.be/ZI_emiRMoxQ`

I. Introduction

JavaScript has become the language for web applications, but JavaScript web applications are often vulnerable to programmer errors and security attacks. Because JavaScript provides extremely functional and dynamic features and high portability with web environments, web developers build web applications mostly in JavaScript these days. However, the characteristics that brought the prevalent uses of JavaScript web applications also introduce difficulties in building correct web applications. The extremely functional and dynamic features make programs hard to write correctly and hard to reason about. Also, the dynamism and portability of web environments make web applications vulnerable to security attacks.

To help JavaScript developers build correct programs, researchers have developed analyzers that support various analysis techniques, but they are not especially designed for usability. SAFE[1] [6] is a scalable analysis framework for ECMAScript, and it has adopted recent research results incrementally. While more analysis techniques make SAFE more featureful, they make the code base of SAFE large and complex, which makes it difficult for new users to understand the code base. TAJS[2] [1] is a dataflow analysis for JavaScript that infers type information and call graphs. Even though it provides a large collection of analysis techniques like SAFE, it does not provide a facility to understand the analysis status during analysis. Similarly for SAFE, it is not easy for a novice user to extend TAJS to experiment with new analysis techniques. WALA[3] [14] was originally developed for pointer analysis of Java programs, and it also supports flow-insensitive pointer analysis of JavaScript programs. Because WALA aims to support analysis of multiple programming languages, the source code repository is gigantic with many packages, which incurs a huge learning curve.

The more advanced analysis techniques are integrated, the more difficult it is for analyzer users to understand the code base. For example, SAFE was first designed to analyze pure JavaScript benchmarks [6]. It provides a default static analyzer based on the abstract interpretation framework, and it performs several preprocessing steps on JavaScript code to address quirky semantics of JavaScript such as the `with` statement [9]. It was then extended to model web application execution environments of various browsers [11] with HTML/DOM tree abstraction, and it supports analysis of interactions between JavaScript code and native functions in platform-specific libraries by using automatic modeling of library functions from API specifications [2]. Recent extensions of SAFE include aggressive integration of soundy [7] analysis. Instead of analyzing the entire concrete behaviors of programs, it supports an analysis of partial programs by using approximate call graphs from WALA [5]. It also utilizes dynamic information statically to focus on specific environments like specific browser versions [12]. The more features are integrated, the more complicated the SAFE code base gets.

In this paper, we present SAFE 2.0[4], a playground for advanced research in JavaScript web applications[5]. We designed it to be light-weight, highly parametric, and modular. SAFE 2.0 has the following main features:

- Pluggability: To help developers experiment with analysis techniques easily, SAFE 2.0 enables analysis sensitivities and even abstract domains to be selected at run time.
- Extensibility: In order for researchers to implement their new ideas easily or to reproduce research achievements from the literature quickly, SAFE 2.0 supports well-designed APIs for adding new phases or options.
- Debuggability: To aid SAFE 2.0 users to understand and reason about analysis results easily, it supports HTML Debugger, which lets users investigate analysis status from browsers. Users can also test their analysis implementation with Test262, the official ECMAScript conformance suite.

[1] https://github.com/sukyoung/safe/tree/SAFE1.0
[2] https://github.com/cs-au-dk/TAJS
[3] http://wala.sourceforge.net/wiki/index.php
[4] https://github.com/sukyoung/safe (BSD license)
[5] We call SAFE "SAFE 1.0" from now on to distinguish it from SAFE 2.0.

978-1-5386-1589-8/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-C.2017.4

```
val commands: List[Command] = List(CmdParse, CmdASTRewrite, CmdCompile, CmdCFGBuild, CmdAnalyze,
                                   CmdBugDetect, CmdHelp)
var phases: List[Phase] = List(Parse, ASTRewrite, Compile, CFGBuild, Analyze, BugDetect, Help)
```

(a) Add a new command and a new phase to the lists of available commands and phases, respectively, in `Safe.scala`

```
case object CmdBugDetect extends CommandObj("bugDetect", CmdAnalyze >> BugDetect)
```

(b) Add the new command by specifying its name and phases in `Command.scala`

```
case object BugDetect extends PhaseObj[(CFG, Int, CallContext), BugDetectConfig, CFG] {
  val name: String = "bugDetector"
  val help: String = "Detect possible bugs in JavaScript source files."
  def defaultConfig: BugDetectConfig = BugDetectConfig()
  val options: List[PhaseOption[BugDetectConfig]] = List(
    ("silent", BoolOption(c => c.silent = true), "messages during analysis are muted.")
  )

  def apply(in: (CFG, Int, CallContext), safeConfig: SafeConfig, config: BugDetectConfig): Try[CFG] = {
    val (cfg, _, _) = in
    // Bug detector implementation here.
    Success(cfg)
  }
}

case class BugDetectConfig(var silent: Boolean = false) extends Config
```

(c) Implement the new phase in `phase/BugDetect.scala`

Fig. 2. Extension of SAFE 2.0 with a bug detector [11]

```
{
  "analyzer": {
    "callsiteSensitivity": 1
  },
  "file": ["wikipedia.org.htm"]
}
```

(a) Specifies selected analysis sensitivity

```
{
  "analyzer": {
    "number": "flat",
    "maxStrSetSize": 10
  },
  "file": ["wikipedia.org.htm"]
}
```

(b) Specifies selected abstract domains

Fig. 1. SAFE 2.0 analysis configuration files in JSON format

II. SAFE 2.0 Features

A. Pluggability

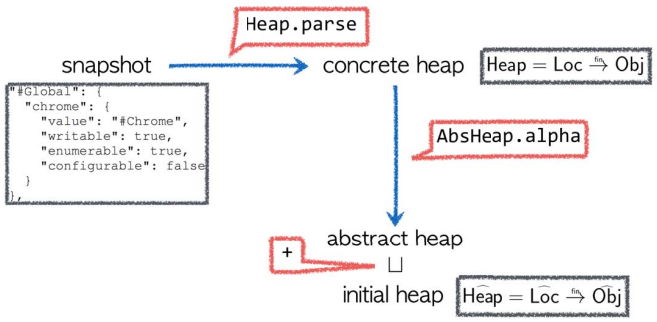
SAFE 2.0 is designed to support selection of analysis sensitivities and abstract domains at run time. A large body of research on program analysis has been designing, developing, and evaluating various analysis techniques like analysis sensitivities and abstract domains. For example, researchers have evaluated the analysis precision and scalability with different n for n-depth loop unrolling or different k for k-CFA, which distinguishes the same function body from its different call sites using k call history [10]. Moreover, various abstract domains for JavaScript string analysis have been proposed such as a regular expression domain [8] and a string automata domain [4]. However, with existing analyzers, it is not easy to configure analysis sensitivities and abstract domains without recompilation of the analyzers.

In order to help analysis researchers experiment with different analysis techniques easily, SAFE 2.0 provides APIs that support pluggable analysis sensitivities and abstract domains. The APIs enable SAFE 2.0 users to select specific analysis techniques. For example, while the default abstract domain for strings in SAFE 2.0 is a string set domain, one can use a regular expression domain or a string automata domain. Because specifying various selection as command-line options multiple times is tedious and error-prone, we provide an easy way to specify them in a configuration file in JSON format. The configuration file shown in Figure 1(a) specifies that it uses the 1-CFA analysis sensitivity, and the one in Figure 1(b) specifies that it uses a flat number domain and a string set domain of maximum size 10.

B. Extensibility

Researchers may devise new analysis techniques or they may want to reproduce analysis results reported in the literature. In either way, depending on which analysis technique is being implemented, one may have to add a new phase or it may suffice to add one option to an existing phase. For example, when a researcher wanted to add a simple symbolic executor to SAFE 1.0, the researcher had to modify many functions in different files without much help from API functions. In order to add a command, a phase, an option, a help message, and an implementation of the new functionality, the researcher had to understand many low-level details of SAFE 1.0.

Based on our own painful experiences, we revamped the structures of the main SAFE 2.0 driver, commands, phases, options, and configurations, and provide well-designed APIs



(a) Use of dynamic heap information to reduce false positives [12]

```
def addSnapshot(st: AbsState,
                snapshot: String): AbsState = {
  val concreteHeap = Heap.parse(snapshot)
  val abstractHeap = AbsHeap.alpha(concreteHeap)
  AbsState(st.heap + abstractHeap, st.context)
}
```

(b) Implement the new technique

```
("snapshot",
 StrOption((c, s) => c.snapshot = Some(s)),
 "analysis with an initial heap generated \
  from a dynamic snapshot(*.json)."),

config.snapshot.map(str =>
  initSt = Initialize.addSnapshot(initSt, str))
```

(c) Add a new option in `Analyze.scala`

Fig. 3. Extension of SAFE 2.0 with dynamic heap information [12]

for adding new commands, phases, and options. For example, when we extended SAFE 2.0 with a bug detector [11], all we had to do was to make small modifications in two files and to implement the bug detector. Figure 2(a) shows how we added a new command `CmdBugDetect` and a new phase `BugDetect` in the main driver file `Safe.scala`. Figure 2(b) presents that we added the new command `CmdBugDetect` with a name `bugDetect` and its phases `CmdAnalyze >> BugDetect`, which denotes that the new phase `BugDetect` is added to the end of the phases of the command `CmdAnalyze`. Then, Figure 2(c) shows the implementation of the bug detector. While the actual implementation is omitted, simply providing its name, help message, configuration, and possible options all in one single file is enough for SAFE 2.0 to plug the information into appropriate places. In this way, users do not need to understand how SAFE 2.0 handles commands, phases, options, and help messages, but they can simply specify what they are for the new command.

Figure 3 illustrates another example extending SAFE 2.0 with dynamic heap information as specified in the literature [12]. The technique is to capture the initial concrete heap for each browser called *snapshot*, to transform it to its corresponding abstract heap, to merge the resulting abstract heap with the default initial heap, and to use the merged heap for the analysis. Thus, the implementation of the technique consists of two parts: a capturing app of dynamic heap, and code that transforms the captured dynamic heap to an abstract heap and merges it with another abstract heap. By using a browser-specific information instead of a sound approximation of all browsers, the technique reduces many false positives from analysis results. When applying the technique to SAFE 2.0, we had to implement only `Heap.parse` that parses a snapshot and constructs a concrete heap in SAFE 2.0, and we could reuse the capturing app in tact and existing SAFE 2.0 APIs for the remaining tasks. As Figure 3(b) shows, for a given snapshot from the capturing app, `Heap.parse` constructs a concrete heap from the snapshot. Then, we can use the existing abstraction function for heaps `AbsHeap.alpha` and the join operation on heaps `+` in SAFE 2.0. As Figure 3(c) shows, one can add a new technique to the existing analysis by simply adding one option.

C. Debuggability

While many programming language environments provide debugging facilities, most static analysis frameworks do not provide such utilities for analysis developers. Because understanding and reasoning about analysis status are extremely difficult, tracking the causes of analysis imprecision is one of the active research topics. However, existing JavaScript analyzers are still in pre-mature stages. SAFE 1.0 provides a console debugger, which allows users to investigate the analysis status during analysis with stepwise execution of the underlying analyzer. It indeed helps SAFE 1.0 users debug the analyzer behaviors, but it lacks documentation and it requires knowledge of the underlying analyzer.

To help SAFE 2.0 users to easily understand and reason about analysis results, it now supports HTML Debugger, which enables users to investigate analysis status from browsers. During analysis, a user can write the current analysis status into an HTML file and investigate it from a browser as illustrated in Figure 4. It shows the current CFG in the middle. Nodes in black lines denote the blocks that are analyzed, those in gray lines denote the blocks not yet being analyzed, and colored nodes denote the blocks that are currently in the worklist of the analyzer. One can toggle whether to show the nodes in the worklist by the menu button on the top right. When a user selects a block from the CFG, the list of the instructions in the block and the state just before analyzing the block are displayed on the left.

SAFE 2.0 has been tested using Test262, the official ECMAScript conformance suite[6], which helped us find and fix bugs in our modeling of builtin functions specified in chapter 15 of the ECMAScript specification [3]. Our regression test suite checks whether the analysis results soundly approximate their corresponding concrete values. It also measures how many tests are being analyzed precisely. This test infrastructure with Test262 will aid SAFE 2.0 users who build new analysis techniques to "test" the soundness and precision of their new analysis implementation.

[6]https://github.com/tc39/test262

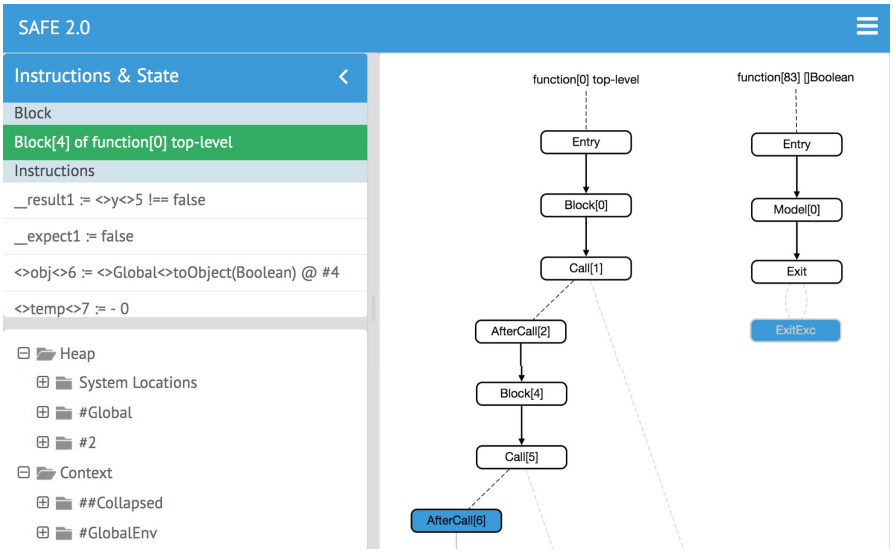


Fig. 4. HTML Debugger

III. Future Directions

Since the release of SAFE 2.0 in early October 2016 [13], we have been extending SAFE 2.0 with various features. Among others, the following features will be supported in near future:

- Easier addition of sensitivities
- More support for abstract domain APIs
- Improvements in HTML Debugger
  We are working on integration of the SAFE 2.0 analyzer and HTML Debugger, which would allow stepwise execution and more interactive debugging from browsers. We plan to use a database for analysis results for scalability.

In addition, we plan to provide basic modeling supports for HTML/DOM tree abstraction and the jQuery library, which has more than 90% of market share. We believe SAFE 2.0 will let us explore more easily the remaining challenges in analysis of JavaScript web applications like event handling, modeling framework, and compositional analysis.

IV. Conclusion

We present SAFE 2.0, a tool that analyzes JavaScript web applications with supports for analyzer users in mind. On top of the well-tested core features of SAFE 2.0 based on the prior experiences from building SAFE 1.0, it also provides mechanisms for analysis developers to experiment with their novel ideas without too much implementation burden. It allows users to specify target phases and options in a declarative manner, which can configure even abstract domains and analysis sensitivities. Analyzer developers can add new analysis techniques into SAFE 2.0 without too much understanding of the implementation details of SAFE 2.0 via its extensible APIs. Finally, SAFE 2.0 provides a debugging support for analysis developers with visualization and interactive investigation of the analysis status. The tool was motivated by the pain the authors themselves experienced with SAFE 1.0, which was greatly relieved by the HTML Debugger. Also, analysis developers can test their new analysis techniques with extensive Test262, the official ECMAScript conformance suite.

References

[1] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *Proceedings of OOPSLA*, 2014.

[2] S. Bae, H. Cho, I. Lim, and S. Ryu. $\text{SAFE}_{\text{WAPI}}$: Web API misuse detector for web applications. In *Proceedings of FSE*. ACM, 2014.

[3] ECMA. ECMA-262: ECMAScript Language Specification. Edition 5.1, 2011.

[4] S.-W. Kim, W. Chin, J. Park, J. Kim, and S. Ryu. Inferring grammatical summaries of string values. In *Proceedings of APLAS*, pages 372–301. Springer International Publishing, 2014.

[5] Y. Ko, H. Lee, J. Dolby, and S. Ryu. Practically tunable static analysis framework for large-scale JavaScript applications. In *Proceedings of ASE*, pages 541–551. IEEE, 2015.

[6] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *Proceedings of FOOL*, 2012.

[7] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Communication of ACM*, 58(2):44–46, 2015.

[8] C. Park, H. Im, and S. Ryu. Precise and scalable static analysis of jQuery using a regular expression domain. In *Proceedings of DLS*, pages 25–36. ACM, 2016.

[9] C. Park, H. Lee, and S. Ryu. All about the `with` statement in JavaScript: Removing `with` statements in JavaScript applications. In *Proceedings of DLS*, pages 73–84. ACM, 2013.

[10] C. Park and S. Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *Proceedings of the European Conference on Object-Oriented Programming*, 2015.

[11] C. Park, S. Won, J. Jin, and S. Ryu. Static analysis of JavaScript web applications in the wild via practical DOM modeling. In *Proceedings of ASE*, 2015.

[12] J. Park, I. Lim, and S. Ryu. Battles with false positives in static analysis of JavaScript web applications in the wild. In *Proceedings of ICSE*, pages 61–70. ACM, 2016.

[13] J. Park, Y. Ryou, and S. Ryu. Safe 2.0 is now available! https://www.mail-archive.com/types-announce@lists.seas.upenn.edu/msg05947.html.

[14] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *Proceedings of PLDI*, 2013.