5-1-2021

A Study on Formal Verification for JavaScript Software

Zachary S. Rowland
University of Dayton

A Study on Formal Verification of JavaScript Software

Honors Thesis

Zachary Rowland

Department: Computer Science

Advisor: Phu H. Phung, Ph.D.

May 2021

A Study on Formal Verification for JavaScript Software

Honors Thesis

Zachary Rowland

Department: Computer Science

Advisor: Phu H. Phung, Ph.D.

May 2021

Abstract

Information security is still a major problem for users of websites and hybrid mobile applications. While many apps and websites come with terms of service agreements between the developer and end user, there is no rigorous mechanism in place to ensure that these agreements are being followed. Formal methods can offer greater confidence that these policies are being followed, but there is currently no widely adopted tool that makes formal methods available for average consumers. After studying the current state-of-the-art in JavaScript policy enforcement and verification, this research proposes several new techniques for applying model checking to JavaScript that strikes a balance of low runtime overhead and fine-grained policy enforcement that other techniques do not achieve.

Acknowledgements

A big thank you to Dr. Phung for advising me throughout the year.

University *of* Dayton

Table of Contents

Introduction

Web technologies have become essential to the proper functioning of our society. Websites and web applications are now the standard method of communicating for business and pleasure. Additionally, many commonly used web services are available as a hybrid mobile application, a smartphone app developed using web technologies. Mobile applications for many commonly used web services such as Gmail, Twitter, and Instagram are implemented as hybrid mobile applications because the majority of the code can be copied from the website implementation reducing the total amount of code that needs to be maintained. Furthermore, the advent of Node.js allowed even server-side code to be written in the same language as the front-end website code.

The language in which the behavior of such web services is implemented is called JavaScript—a dynamic and permissive language that was not developed to tackle the security challenges it faces today. Its lack of security features and error checking makes it easy for the developer to overlook programming mistakes. Its complex and ill-defined specification makes it difficult for the consumer to read and analyze. JavaScript and web security have been the subject of much research since the language's inception and the growth of the web, but no definitive solution to JavaScript's security problems has been implemented on a large scale.

The current security standards for handling sensitive information on the web is the same as standards for handling physical property. Websites and hybrid mobile apps are expected to come with a privacy policy, an agreement between the software developer

and the end user about how the user's information will be used. The privacy policy is enforced with threat of legal action upon violation. This security standard offers very little control to users over their own private information and device usage because legal restrictions can be and frequently are broken with little to no repercussions.

Formal methods of software verification and enforcement offer promising tools for users to gain some level of confidence in the security of the software they use. But currently, the study and implementation of robust formal methods is confined to academic research and big industry. The aim of this work is to study the applicability of formal methods to JavaScript with the eventual goal of integrating formal methods into a tool for user-specified policy enforcement and verification. Additionally, several methods are proposed for providing simplistic model-checking support for JavaScript. There is little research studying the integration of model-checking with the web environment since it is traditionally studied in the context of embedded systems and electronic circuits. However, model-checking has the potential to verify simple security policies relevant to end-users.

This thesis is being conducted within the Intelligent Systems Security Lab, whose research goals include offering software security and privacy solutions to average users. This thesis provides a survey of formal verification tools and techniques for JavaScript among other languages and discusses ways that a specific verification technique called model checking can be implemented for JavaScript.

Formal Methods Overview

Formal methods or formal verification is the use of mathematical modeling and reasoning to prove that software always exhibits correct intended behavior and never unintended behavior. Many software insecurities arise from a mismatch between the behavior that developers *expect* the software to exhibit and the actual exhibited behavior of the final product. These mismatches can arise from either under-specification or mistakes in the implementation. Under-specification occurs when the programmer does not know the exact intended behavior of a software system. Often a programmer will not realize that certain behavior is undesirable until the program exhibits that behavior. This could cause the programmer to make arbitrary implementation choices in situations where the exact behavior of the system is assumed to be unimportant. Implementation mistakes arise when the programmer understands the intended behavior, but fails to write code that properly exhibits this behavior. This is often caused by a lack of understanding or assumptions made about the programming language's proper semantics.

The purpose of Formal Methods is to reduce or eliminate both of these sources of software mistakes. Under-specification can be avoided by using formal specification languages that require every possible input to the program to be properly considered, and implementation mistakes can be avoided by mathematically connecting the concrete program to an abstract specification. Formal methods achieve safety guarantees by relying on precise mathematical modeling and proof techniques. This requires traditionally informal concepts such as a "program" or "safety" to be represented in

formal, machine-readable language. Any system of formal software verification has three fundamental components:

1. A formal language or modeling system for describing software programs.
2. A formal language for expressing *properties* of programs.
3. An algorithm that evaluates a program model against a property expressed in the formal property language.

Since software programs are fundamentally just complex state transition systems, they are usually modeled as such. A transition system describes how a system in a given "current configuration" (i.e. current state), can evolve over time into other configurations via a set of predefined transitions. In a typical imperative programming language, the state of the program consists of the values assumed by all of its variables or the data stored in relevant sections of memory. The transitions would be the program statements themselves that describe how the state is modified over the course of an execution.

Program properties, or policies, can be expressed in one of several ways depending on the verification method being used. Like programs, a property can simply be a transition system that defines the valid ways that the program is allowed to transition. However, properties can also be expressed in propositional or first-order logic, temporal logics, or as a set of reachable states among other methods.

Proof-based Verification

Proof-based verification is one of the two major branches of formal verification techniques. A *proof-based verification* technique is any technique where the program and

properties are represented as logical formulas. Verification under a proof-based modeling system means performing a logical deduction starting from the program's formulas to the desired property.

The most commonly used proof-based verification method for imperative programs is called Hoare Logic which encodes a program as a set of precondition-statement-postcondition triples. A *Hoare triple* is a statement of the form $\{\phi\}C\{\psi\}$ where C is a program and ϕ and ψ are logical formulas that express properties about the program state. The statement $\{\phi\}C\{\psi\}$ means "if condition ϕ holds before program C is executed, then condition ψ will hold after C is executed." For example, if `skip` represents a program that performs no action, then any Hoare triple $\{\phi\}skip\{\phi\}$ will be true. As another example, suppose `x := e;` is a program that evaluates an expression `e` and stores the result in a variable `x`, then $\{\phi[x \to e]\}x \coloneqq e;\{\phi\}$ will be true ($\phi[x \to e]$ represents the formula ϕ with all occurrences of x replaced with e).

In Hoare Logic, a property is a precondition-postcondition pair (ϕ, ψ), and a proof that a program P satisfies the property is a derivation of the statement $\{\phi\}P\{\psi\}$. This derivation is constructed by breaking down P into smaller programs, constructing proofs for each subprogram, and combining the resulting proofs into a single program for P. For example, suppose $P = s_1; s_2$ is a program that executes the two statements s_1 and s_2 in sequence. Then we could derive $\{\phi\}s_1; s_2\{\psi\}$ if we had Hoare triples $\{\phi\}s_1\{\nu\}$ and $\{\nu\}s_2\{\psi\}$ where ν is some intermediate condition.

Proof-based verification offers the benefits of being direct and flexible. Programs don't need to undergo significant translation in order to reason about them. Hoare logic can directly reason about common programming constructs such as if-statements and while-loops which eliminates errors from being introduced during translation. However, proof-based verification is also difficult to automate. As with proofs in other areas of math, it is not trivial to determine how to combine the atomic propositions to reach the desired conclusion.

Model Checking

The other major branch of formal verification is *model-based verification* where the program is transformed into some abstract model (usually a state-transition system) before verification. *Model Checking* is a particular method of model-based verification, but it is general enough to be representative of the entire model-based strategy, so this thesis will largely omit discussion of other model-based methods.

Model Checking is a formal verification system that verifies *temporal properties* against a program represented as a *finite-state transition system*. There is no single standard way to encode programs as transition systems. One such encoding is called a Kripke structure, a triple (S, R, I) where S is a set of program states and I is a mapping from states to sets of atomic propositions. $R \subseteq S \times S$ is the transition relation that specifies which states the program can evolve into from the current state. For deterministic programs, R can be a function $S \to S$. A program can also be modeled as a *labeled transition system* $(S, Act, \to)$ where S is the set of states, Act is a set of *actions*

that can be taken by a program. The arrow $\rightarrow \subseteq S \times Act \times S$ is a three-way relation that describes which states evolve into which other states when some action is performed.

The power of model checkers comes from their ability to verify policies written in a temporal logic. A *temporal property* is a restriction on the order in which states can be reached. A simple example of a temporal property is a *no-send-after-read property* which says that if the program performs some "read" action (such as fetching information from the user's photo album or contacts list), then the program is not allowed to subsequently perform a "send" action (such as transmitting data to a remote server). More formally a temporal property restricts which traces a program is allowed to step through. A valid *trace* or *path* is an infinite sequence of states $s_1, s_2, \ldots$ such that for each pair of adjacent states s_i, s_{i+1}, there is a valid transition from s_i to s_{i+1} in the transition relation.

Temporal properties can be expressed in two main formalisms. The first is *linear-temporal logic* (LTL), an extension of first-order logic that includes constructs for reasoning about changing properties over time. These two constructs are the X and U operators. If ϕ and ψ are LTL formulas, then $X\ \psi$ means ψ is true for every possible *next* state of the system, and $\phi\ U\ \psi$ means that ϕ remains true *at least until* the system reaches a state where ψ is true. Other temporal operators can be defined in terms of these two constructs such as $G\ \psi$ meaning "ψ will always (globally) be true" and $F\ \psi$ meaning "ψ is guaranteed to eventually be true after a finite number of transitions". Linear temporal logic is the property language used by the SPIN model checker discussed below.

The other temporal logic is *computation tree logic* (CTL) which introduces trace quantifiers A (for all traces) and E (there exists a trace) in addition to the X and U

operators. These quantifiers allow for expressing that *it is possible* for some property to be true, but the execution of the program might not follow the trace that holds that property. In CTL, a trace quantifier or a temporal operator cannot stand by itself. CTL can express properties about "every state in every trace" or "every state in at least one trace", but does not allow for reasoning about "every trace" directly. CTL is the temporal language used by the NuSMV model checker.

Because there are many languages that may need model checking, popular general-purpose model checkers generally implement model checking on a simpler custom language called a modeling language. To use a general-purpose model checker, a developer would need to translate their program from the language in which it was written into the modeling language of the model checker. SPIN and NuSMV are both general purpose model checkers that follow this design philosophy. SPIN (which stands for Simple Promela Interpreter) operates on C-inspired meta-language called Promela. SPIN is useful for verifying properties of concurrent programs because of Promela's built-in support for processes. The input language for NuSMV is much closer to a direct specification language for finite-state machines which makes NuSMV less amenable to software-verification than SPIN. However, NuSMV makes up for it in speed, support for CTL logic in addition to LTL, and more advanced model checking techniques such as bounded model checking.

There are, however, model checking tools available for widely used practical programming languages. CPAchecker is a hybrid tool for model-checking and dataflow analysis (discussed below) that operates directly on the C language. Policies are specified

using LTL or a policy automaton. Java PathFinder is a similar tool for Java. These tools must deal with aspects of real programming languages such as floating-point arithmetic, dynamically-sized data structures, and pointer manipulation that most general-purpose model checkers intentionally avoid. Consequently, they run slower and produce more confusing failure analysis.

State-space Exploration Beyond Model Checking

While model checking encapsulates most other verification techniques centered around traversing the program's state-space, the simplifications made by other techniques are worth mentioning. These other techniques are not applicable for verifying temporal properties, only for determining reachability of a state or set of states from the program's starting state.

A *dataflow analysis* requires states to be expressed as a tuple of values $(x_1, x_2, \dots, x_n)$ which represent variable bindings. Dataflow simplifies the state-space search by eliminating information about how the fields of the state relate to each other. Specifically, if a variable x_i assumes a value v_i in some reachable state s_1, and a variable x_j assumes v_j in some other reachable state s_2, then a dataflow analysis will assume that there is some reachable state where both x_i and x_j assume those values simultaneously. Instead of maintaining a set of reachable states, the algorithm only needs to maintain a much smaller set of reachable *values* for each variable. This dodges the state-space explosion problem that model checking has because the amount of memory needed to

store the set of reachable states is now linear instead of exponential in the number of variables.

Symbolic execution is more directly geared toward analyzing software programs than model checking. A symbolic executor will interpret a program by initializing program variables to symbolic values. When operations are performed on these symbolic values, the computations that would be performed are simply saved in the form of a mathematical expression. When the symbolic execution is complete, state-reachability properties can be verified by analyzing the resulting expressions for each program variable.

Information Flow

With regard to information privacy in particular, policies often need to talk about the history of a piece of data, not just its contents. An information-flow policy restricts the actions a program can perform on a piece of data based on the history of function calls and computations that generated the data. For example, the standard web environment provides a JavaScript API function `getcurrentposition` that asks the browser to determine the current geolocation coordinates of the device using GPS or other means. Apps and websites often passively collect geolocation coordinates just for information gathering purposes. The user might want to block this passive collection while allowing the coordinates to be used for desired tasks such as navigation in a Maps application.

Information-flow properties of a program can be verified statically using a form of abstract interpretation, but currently it is more common to guarantee safety by enforcing information-flow such policies at runtime. Enforcement usually involves modification of the runtime environment or heavy instrumentation of the program code. While information-flow policies are very expressive and relevant to most users, runtime enforcement usually entails significant overhead. The most common way of enforcing information-flow is a technique called *secure multi-execution* which, as the name implies, requires executing the program multiple times concurrently causing the program to run at least several times slower.

The significant enforcement overhead is required since information flow policies are, in general, more *fine-grained* than temporal policies. Enforcing a no-send-after-read policy is one way of making sure information does not leak out of a program, but since *any* send after *any* read is blocked, plenty of benign "send" actions will also be blocked despite not containing any sensitive information. Information flow policies can often express more accurately the precise behaviors that the user wants to allow and block without over- or undercompensating.

JavaScript & Web Security Survey

Here, we examine the current state-of-the-art verification tools for JavaScript that make use of the formal verification techniques discussed above. Most work in JavaScript language-based security can be categorized into one of three categories. Formalizing the semantics of JavaScript means mathematically describing the precise procedure for the

execution of a JavaScript program which has been hitherto informal. Proper formalization of the language semantics provides the groundwork for the other two methods of static policy verification and dynamic policy enforcement.

Formalizing JavaScript Semantics

Any solid verification framework must be based on a precise, formal understanding of the execution of a program. Unfortunately, JavaScript interpreters were built ad-hoc, so their behavior is not fully understood. JavaScript's runtime semantics are described informally as the ECMAScript specification, but this specification is lengthy and includes ambiguities. Therefore, the past two decades have seen a number of attempts to formalize the semantics of JavaScript. In 2008, Maffeis et. al. defined the first operational semantics for JavaScript. The first mechanized semantics was JSCert (Bodin, 2014), a formalization of the JavaScript semantics in the Coq theorem-prover. The Coq semantics have the benefit of being mechanized making them easier to work with without making mistakes. However, using the semantics in theorem proving is still only semi-automatic and requires a lot of human intervention. In 2015, Park et. al. modeled the JavaScript semantics in the K-semantics framework. The K system automatically generates a parser and model-checker for the semantics.

JavaScript is a very large language and formalizing the entire specification is a huge undertaking. Furthermore, the formalized semantics, while being complete, would not be tractable enough to implement in practice. These concerns prompted Guha et. al.

to take a different approach: defining a smaller language called lambda-JS with fewer constructs and translating JavaScript into lambda-JS via a "desugaring process".

Static Verification

Automatic program verification did not start being developed until later. A major step forward in automatic JavaScript verification is the symbolic execution tool JaVerT (JavaScript Verification Tool) (Fragoso Santos, 2019). JaVerT allows users to verify precondition-postcondition properties similar to how one would with a proof-based verification system. JaVerT is built on separation logic, an extension of Hoare Logic that enables reasoning about programs that manipulate pointer values and more complex data structures. JaVerT allows verification to be compositional—proven properties of individual functions of a program can be combined into properties of the program as a whole.

The verification of individual functions of a program is an important part of verification, however the input-output policy specification limits JaVerT's ability to handle programs that, in theory, are designed to run indefinitely. Unfortunately, most web programs are designed to handle various events and callbacks such as mouse clicks or displaying information in real time which requires an execution thread to be alive indefinitely.

Dynamic Policy Enforcement

Infrastructure for analyzing information flow through JavaScript programs has been implemented by JSFlow (Hedin, 2014) and FlowFox (De Groef, 2012). JSFlow is implemented as an extension to the Firefox web browser. FlowFox is implemented as a modified Firefox browser.

Phung et. al. (2009) proposed a policy enforcement technique called lightweight self-protecting JavaScript where a JavaScript program is modified before runtime to monitor calls to built-in API functions. This technique exploits the fact that JavaScript allows any code to redefine global variables. Since applications are given access to user's data via JavaScript API functions bound to variables, these API functions can be "rewritten" to implement a desired security policy.

The desired policy is enforced via an *inlining process* that occurs after the sensitive API functions are defined. Suppose the policy requires restricting access to the function `getPosition` in the `nav.geoloc` object. Before any untrusted code is executed, a reference to the original `getPosition` function is saved to be accessed later by the enforcement code. The enforcement code is then defined as a JavaScript function and assigned to the global variable `nav.geoloc.getPosition` overwriting the reference to the original function. Upon invocation, this new function will check if the calling the original sensitive API function will cause a policy violation. If not, execution proceeds with the original invocation, otherwise the original invocation is blocked.

The benefits of self-protecting JavaScript are the ease of implementation and small runtime overhead. The inlining code can be implemented in JavaScript itself and

injected into a website or hybrid mobile application by an additional script inclusion. Unlike information flow, runtime overhead only occurs at the beginning of the execution (to load the enforcement script) and when the untrusted code attempts to call a sensitive API function.

However, the policies that are enforceable by self-protecting JavaScript are limited when compared to other methods. No monitoring of the behavior of the untrusted code is done apart from calls to API functions. This means information flow policies cannot be enforced directly as it is not possible to track information as it is passed between variables. The primary set of policies that can be handled are temporal policies on the set of monitored API functions, which prompts the use of model checking to verify the soundness of the enforcement. The next section discusses several strategies for integrating model checking with JavaScript.

Proposed JavaScript Model Checking Framework

One of the main goals of this thesis was to develop a method of implementing model checking for JavaScript. To our knowledge, there is no widely used existing tool for validating JavaScript code using model checking. There are also many security policies relevant to end-users that can be expressed as temporal properties. Users may want to limit the frequency of certain actions to combat passive information gathering or restrict reading and sending personal information. Temporal properties are also easily enforced using lightweight self-protecting JavaScript, so interoperability between the policy enforcement mechanism and the static policy validation method could be possible.

Since constructing a model checker from scratch would be time consuming, instead a series of translators were built from JavaScript into other languages with dedicated model checking tools. The following subsections discuss implementations of translators to C and NuSMV as well as a strategy for encoding JavaScript programs and temporal policies in Datalog.

Building such a translator requires overcoming two major hurdles. The first challenge is deciding how much of the JavaScript syntax and semantics should be supported by the translator. Enough of the language needs to be supported to be useful, but supporting the entire language would be infeasible. The second challenge is developing a method of formal abstraction for the translator, or a set of formal guidelines that detail the amount and kinds of details that are preserved and discarded during the translation.

JavaScript to C Translation

The C language is a good candidate for a target language because of its variety of matured analysis tools such as CPAchecker. It is also much more expressive than general purpose modeling languages such as Promela which were not designed to be executed as normal programs. The C language is powerful enough to emulate key JavaScript features like dynamic typing and IEEE-754 floating-point arithmetic. To take advantage of the benefits of C, we have developed a prototype translator called JS2C which preserves the semantics of the original JavaScript program as much as possible. More specific information including code snippets can be found in Appendix A.

Dynamic typing is emulated in C by representing all possible JavaScript values inside a structure called `JSVar` which contains type information in addition to the value itself. Values of primitive types such as `number` and `boolean` in JavaScript are easily mapped to instances of `JSVar` in C. Likewise, all JavaScript operators and functions are mapped to C functions that operate on `JSVar` instances. Currently, only non-dynamic primitive types are supported, but support for JavaScript objects and strings could be supported in the future using the heap and careful memory management. First-class functions could possibly be implemented using function pointers. More advanced features like dynamic code generation and the `eval` construct might not be feasibly supported.

JavaScript to NuSMV Translation

While translation to C would be more direct, making simplifications during the translation is in many cases desirable because reducing complexity reduces the likelihood of introducing errors. Additionally, if the code being analyzed is intended to be policy enforcement code such as the code injected by self-protecting JavaScript, then support for more complex JavaScript semantics might not even be necessary. Enforcement code should only be constructed using simple language constructs anyway to avoid complexity. The prototype JS2NuSMV translates JavaScript programs that follow precise formatting guidelines into a representative NuSMV module.

Additionally, our code instruments the original JavaScript to bring the semantics of the original program closer to the semantics of the NuSMV language. This is

accomplished by injecting calls to custom JavaScript to make sure that type checking or arithmetic operators are performed correctly. For example, a call to a function `opADD` will be substituted for each use of JavaScript's addition operator. This function is less-permissive than the addition operator performing only 32-bit signed integer arithmetic and throwing an exception for invalid arguments. The modified JavaScript program could then theoretically be deployed instead of the original written JavaScript for extra safety guarantees at the cost of some runtime overhead.

This method provides the benefit of flexibility, but places heavy restrictions on what constructs are allowed in a JavaScript input program. Features like dynamic typing, floating-point arithmetic, and function calls are often difficult to implement in modeling languages, and implementing dynamic code generation and eval seems practically impossible.

Datalog Implementation

Implementing a translation from one syntax to another can be confusing and inelegant. Another option is to encode information about JavaScript programs in Datalog, a logic-programming language based on the concepts of facts and rules. A *fact* is a construct of the form `predicate(obj1, obj2, …)` that specifies some relationship between the objects in the parentheses. A *rule* specifies how to derive a fact given a set of other facts. Given a set of initial facts and rules, Datalog will find all facts that can be derived by repeated application of the rules.

Information about the reachability of program states can be encoded using a fact `path(s1, s2)` which means that there exists a sequence of valid transitions from state `s1` to state `s2`. The `path` predicate satisfies a transitive property that can be encoded as the rule `path(s1, s3) :- path(s1, s2), path(s2, s3).`

The `path` predicate can be used to verify some temporal policies like the no-send-after-read policy. First, the program statements need to be encoded as corresponding rules involving `path`. Suppose in state $s1$, the program has just performed a "read" action while in $s2$, the program has just performed a "send" action. Then the policy is violated if the Datalog system is able to derive any such rule `path(s1, s2).` Appendix C includes an example of how a specific program can be encoded and verified.

Automaton Encoding in JavaScript

One of the major challenges of formal verification is properly abstracting a program written in a practical programming language such as JavaScript into a mathematical model. Verifying properties on a general program requires dealing with the complex semantics of the language in which it was written. However, if we turn our focus away from policy verification to policy enforcement, the task becomes easier. This section describes a method of generating self-protecting JavaScript enforcement code that is correct with reasonable certainty.

Suppose that we want to enforce a temporal property that restricts access to a set of global API functions. We will express the property as a transition system $A = (S, M, \delta)$ where S is a set of states, M is the set of sensitive API functions that need to be

monitored, and $\delta: S \times M \rightarrow S \cup \{\text{undefined}\}$ is a partial function mapping state-transition pairs to next states. A sequence of method calls $m_1, m_2, \ldots, m_n$ starting from an initial state s_i is considered safe if $\delta(\cdots \delta(\delta(s_i, m_1), m_2) \cdots, m_n) \neq \text{undefined}$.

We can enforce this temporal property onto untrusted code by using self-protecting JavaScript. The enforcement code will keep track of the state of the program as the untrusted code calls API functions. If the untrusted code attempts to take a transition that does not exist, the enforcement code will block the call so that the security state remains defined. Each call to an API function m must be replaced with an enforcement method that performs the following:

1. $s' \leftarrow \delta(s, m)$;
2. $if\,(s' \neq \text{undefined})\,\{\, s \leftarrow\ s';\ m(\text{arguments});\ \}$

We can implement an automaton in JavaScript directly. If the number of security states and monitored methods is finite and tractable, the states and transitions can be encoded in JavaScript as string values and the transition function δ can be encoded as an object `delta` where `delta["s"]["m"]` evaluates to either a new state or `undefined`. For example, a reduce-reset policy with six states and two transition functions is easily representable:

```
var reduceResetAutomaton = {
    "initialstate": "5",
    "transitions": {
        "0": {"reset": "5"},
        "1": {"reduce": "0", "reset": "5"},
        "2": {"reduce": "1", "reset": "5"},
        "3": {"reduce": "2", "reset": "5"},
        "4": {"reduce": "3", "reset": "5"},
        "5": {"reduce": "4", "reset": "5"}
```

```
    }
};
```

The benefit of encoding the security policy directly in JavaScript is that the same monitor code can be used for all security policies. This means that a proof of correctness of the enforcement mechanism would come down to proving properties of a small amount of fixed JavaScript code instead of general JavaScript programs.

Conclusion

A step toward the development of lightweight user-centered verification of JavaScript is developing model checking support for JavaScript software. This paper presented implementations of translators from JavaScript into C and NuSMV which demonstrate the usefulness and feasibility applying model checking. Future work should include the implementation of a full JavaScript model checking pipeline. Ideally, a single tool would handle both the abstraction of JavaScript programs into models and the model checking itself. However, the abstraction and validation processes could remain separate components. Significant effort would need to be put into formalizing the abstraction of JavaScript programs based on a mechanized JavaScript semantics. Additionally, there will almost certainly be aspects of JavaScript that a model checking tool would not be able to properly handle, so a rigorous specification of allowable input programs must also be developed. Such a tool could become the basis for new information security standards for the web based on demonstrable proof of policy adherence instead of trust of adherence to legal obligations.

While significant progress has been made on language-based web security, our ability to analyze the security of JavaScript and provide safety guarantees is still limited. Static analysis tools like JaVerT are big steps forward, but still only have limited uses and adoption. Information flow enforcement tools like JSFlow and FlowFox are good demonstrations of the feasibility of information flow analysis for JavaScript. However, significant runtime overhead and incompatibility with some webpages hinder their potential for practical use. Lightweight runtime monitoring techniques such as Phung's self-protecting JavaScript are most likely able to provide the desired balance between policy expressivity and efficiency. However, confidence of the soundness of the implementation will only come with a close comparison with formal JavaScript semantics. and a formalization of the enforcement code needs to be done to gain more confidence that the enforcement is sound and tamper-proof. As technology improves and more research is conducted, a workable solution to JavaScript security should come into focus sometime in the near future.

References

Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., and Smith, G. A trusted mechanized JavaScript specification. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* (2014), 87-100.

Bugliesi, M., Calzavara, S., and Focardi, R. Formal methods for web security. *Journal of Logical and Algebraic Methods in Programming 87* (2017).

De Groef, W., Devriese, D., Nikiforakis, N., and Piessens, F. Flowfox : A web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, Association for Computing Machinery, p. 748-759.

Fragoso Santos, J., Maksimovic, P., Sampaio, G., and Gardner, P. JaVerT 2.0: compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1-31.

Guha, A., Saftoiu, C., and Krishnamurthi, S. The essence of javascript. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 6183 LNCS (2010), 126-150.

Hedin, D., Birgisson, A., Bello, L., and Sabelfeld, A. JSFlow: Tracking information flow in JavaScript and its APIs. *Proceedings of the ACM Symposium on Applied Computing* (2014), 1663-1671.

Maffeis, S., Mitchell, J. C., and Taly, A. An Operational Semantics for JavaScript. Advances in Parallel Computing 13, C (2008), 63-70.

Park, D., Stefnescu, A., and Rosu, G. KJS: A complete formal semantics of JavaScript. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2015-June (2015), 346-356.

Phung, P. H., Reddy, R. S., Cap, S., Pierce, A., Mohanty, A., and Sridhar, M. A multi-party, fine-grained permission and policy enforcement framework for hybrid mobile applications. Journal of Computer Security 28, 3 (2020), 375-404.

Phung, P. H., Sands, D., and Chudnov, A. Lightweight self-protecting JavaScript. Proceedings of the 4th International Symposium on Information, Computer and Communications Security, ASI-ACCS'09 (2009), 47-60.

Pupo, A. L. S., Nicolay, J., and Boix, E. G. Guardia: Specification and enforcement of JavaScript security policies without VM moditications. ACM International Conference Proceeding Series (2018).

Taly, A., Erlingsson, U., Mitchell, J. C., Miller, M. S., and Nagra, J. Automated analysis of security-critical JavaScript APIs. Proceedings – IEEE Symposium on Security and Privacy (2011), 363-378.

Appendix A: JS2C Implementation Details

JS2C prepends its output C file with a prelude that emulates JavaScript's semantics with C structures and functions. Below is a sample of the constructs included.

```
// Numbers, Booleans, and undefined types are supported
enum JSType {JSnumber, JSboolean, JSundefined};

typedef union {
    double asNum;
    long asBool;
    unsigned long asHex;
} JSVal;

// A JavaScript type coupled with data
typedef struct {
    enum JSType type;
    JSVal val;
} JSVar;

// Common values are represented using macros
#define jsTrue ((JSVar){JSboolean, {.asBool = 1}})

// Implicit operations defined in the ECMAScript specification
// are implemented as C functions.
JSVar ToBoolean(JSVar v) {
    return ((v.type == JSboolean) ? v :
            (v.type == JSundefined) ? jsFalse :
            (v.val.asNum == 0.0) ? jsFalse :
            (ISNAN(v.val.asHex)) ? jsFalse :
            jsTrue);
}

// The semantics of JavaScript operators are emulated as C functions.
JSVar jsAND(JSVar x, JSVar y) {
    return (ToBoolean(x).val.asBool) ? y : x;
}
```

Appendix B: JS2NuSMV Implementation Details

JS2NuSMV prepends a prelude to the modified JavaScript program that redefines common operators. Below is an example of the redefinition of the JavaScript addition "+" operator to only perform signed 32-bit integer arithmetic:

```
function opADD(x, y) {
    if (!Number.isInteger(x) || !Number.isInteger(y))
        throw "Arguments must be integers";
    var result = x + y;
    if (result >= 2**31)
        result -= 2**32;
    else if (result < -(2**31))
        result += 2**32;
    return result;
}
```

Appendix C: Datalog Verification Details

The code below is a simple representation of enforcement code that implements a no-send-after-read policy. The functions `trySend` and `tryRead` represent public API functions exposed to the untrusted code while the functions `send` and `read` represent the sensitive API functions that need to be monitored. The enforcement code acts as a mediator between the untrusted and sensitive code bases.

```
var canSend = true;

function trySend() {
    if (canSend) send();
}

function tryRead() {
    canSend = false;
    read();
}
```

This enforcement code is modeled as the following Datalog program. The program state is represented as a pair (c, a) where $c \in \{\text{true}, \text{false}\}$ represents the value of the `canSend` variable and $a \in \{\text{none}, \text{send}, \text{read}\}$ is the last sensitive API function that was called. The predicate `path(c1, a1, c2, a2)` means there is a path from state $(c1, a1)$ to state $(c2, a2)$. The functions `trySend` and `tryRead` are each translated into a set of rules that describe the action performed and the value of `canSend` after calling the function.

```
% start state is reachable
path(true, none, true, none). canSend

% models the behavior of trySend
path(true, A, true, send) :- path(true, none, true, A).
path(false, A, false, none) :- path(true, none, false, A).
```

```
% models the behavior of tryRead
path(C, A, false, read) :- path(true, none, C, A).

% transitive property
path(C1, A1, C3, A3) :- path(C1, A1, C2, A2),
                        path(C2, A2, C3, A3).
```

The no-send-after-read policy is satisfied if and only if the query "`path(C1, read, C2, send)?`" returns any facts. The presence of such a fact would mean that that a send was performed after a read, and that these two states are reachable from the starting state.

Appendix D: Automaton-based Encoding Details

A temporal policy can be enforced in JavaScript by using self-protecting JavaScript to inject a security automaton into a program. The security automaton is represented as an object in JSON format. Below is an example of a reduce-reset automaton that allows a “reduce” action to be performed a maximum of five consecutive times before a “reset” action is taken.

```
var reduceResetAutomaton = {
    "initialstate": "5",
    "transitions": {
        "0": {"reset": "5"},
        "1": {"reduce": "0", "reset": "5"},
        "2": {"reduce": "1", "reset": "5"},
        "3": {"reduce": "2", "reset": "5"},
        "4": {"reduce": "3", "reset": "5"},
        "5": {"reduce": "4", "reset": "5"}
    }
};
```

The abstract transitions need to be associated with method calls in the JavaScript program. This can be done by defining a `transitionbindings` object. When the untrusted code calls a sensitive API function, the corresponding abstract transition is retrieved from this object.

```
var transitionbindings = [
    {"object": api, "method": "sendSMS", "transition": "reduce"},
    {"object": window, "method": "handlereset", "transition": "reset"}
];
```

Finally, at the beginning of the program’s execution, the API functions need to be rewritten as described by self-protecting JavaScript. This code rewrites each JavaScript method in the `transitionbindings`. The new self-protecting methods transition the abstract

state in parallel with the execution of the concrete program and block sensitive API calls if the corresponding abstract transition is not present.

```
function enforceAutomaton (automaton, transitionbindings) {
    var state = automaton.initialstate;

    var getMonitorFunction = function (transition) {
        return function (obj, func, args) {
            newstate = automaton.transitions[state][transition];
            if (newstate === undefined
            || !(newstate in automaton.transitions)) {
                console.log("MONITOR: transition not allowed.");
                return;
            }
            state = newstate;
            console.log("MONITOR: new state is " + state);
            func.apply(obj, args);
        };
    };

    for (binding of transitionbindings) {
        intercept(binding.object, binding.method,
                  getMonitorFunction(binding.transition));
    }
}
```