Applying Meta-Functions for Improving JavaScript Code Performance

Ricardo Medel, Alexis Ferreyra, Nestor Navaro, and Emanuel Ravera

Departamento de Ingeniería en Sistemas de Información
Universidad Tecnológica Nacional - Facultad Regional Córdoba
Córdoba, Argentina
{ricardo.h.medel,alexis.ferreyra,nestornav,ravera.emanuel}@gmail.com

Abstract. In recent years, the expansion of the World Wide Web, and web runtimes in particular, to all kind of devices has rendered the JavaScript performance in a hot topic. Several approaches to improve the performance of JavaScript applications have been tried by the industrial and research communities. In this paper we review the most popular approaches and propose a novel solution based on meta-programming and source code rewriting. The preliminary results of our experiments are very promising, although more studies are required to know to what extent this approach can improve the performance of real-life JavaScript programs.

Keywords: Performance, Meta-programming, Rewriting, JavaScript, Macros, Compiler

1 Introduction

Improving JavaScript performance have been a hot topic in recent years. The ubiquity of the World Wide Web, and web runtimes in particular, across all kind of devices in addition to the increasing complexity of applications written in JavaScript have drawn attention of the industrial and research communities towards the improvement of JavaScript performance and security.

In this paper we focus on the performance aspects of this problem. Here we propose a novel solution based on meta-programming and code rewriting, show some preliminary results based on a small set of experiments, and then review other approaches and compare them with our proposal.

Most of the work done to boost the performance of JavaScript programs has been focused on improving the compiling & execution infrastructure. Such approach is implemented, for example, by the Safari SquirrelFish runtime [10], the just in time (JIT) compiler Google V8 [17], and also applied on the trace-based optimizations [9] by the Mozilla team. A different approach, also tried by Mozilla, is implemented in asm.js [13]. This language is a JavaScript subset from which highly efficient code can be generated on the fly. Moreover, Intel improves the performance by adding new extensions to the language, like SIMD [5] (Single

Instruction - Multiple Data) instructions or support for parallel execution [12] of certain APIs (Application Programming Interfaces).

Our work, on the other hand, explores the feasibility of improving JavaScript performance by using source code rewriting techniques at compile time, well before the source code reaches the JavaScript runtime. We found several cases where performance is highly sensitive to the way in which the programmer writes the code. For example, as we show in Sect. 2, using the `for-in` statement to iterate an array object is at least 25 times slower than using a standard C-style `for` to iterate the same array. Even more remarkable, different APIs generating the same, or semantically similar, results can show a big difference in execution times. For example, using the API `document.getElementByTagName` to find elements in the DOM can be 200 times faster than using the API `document.querySelectorAll`.

Therefore, we propose to improve the performance of JavaScript programs by analysing this kind of cases, which are out of reach of JIT compilers and runtimes. To test our hypothesis we initially identified a number of patterns amenable of being optimized, and then implemented a set of simple experiments to check the performance gains. Once we shown that replacing slow patterns with the fast ones provided measurable performance benefits, we designed and implemented a general tool to easily automate this refactoring. The tool, named PumaScript [29], is a JavaScript dialect extended with program introspection and rewriting capabilities. Finally, by using PumaScript we were able of using meta-functions to automatically rewrite slow code with faster, semantically equivalent code.

In the following section we describe in detail the patterns we discovered and the code that can replace them. The third section is devoted to explain the implementation of PumaScript and how to use meta-functions to rewrite slow code. In the fourth section we propose to apply our framework to security issues. Then, in the fifth section, we review in more detail the related work, analysing different approaches to solve the same problem and showing their advantages and disadvantages. Finally, we close this paper by reviewing our results and proposing future lines of work.

2 Analysis of JavaScript Patterns

In this section we compare the performance of several code patterns with semantically similar code but with different execution time. We also analyse and propose solutions for some of the cases, where the semantics is not exactly the same but the expected effects are similar.

Currently, we have identified five JavaScript patterns that are semantically equivalent but with very different execution time. The patterns are shown in Table 1. The first column of the table gives a number to the pattern, while the second column describes the pattern in words. The third column of the table is divided in two rows: the upper row shows the original, slower code, while the lower row shows faster, semantically equivalent code.

#	**Pattern description**	**Original & Improved Code**
1	Native selector by ID vs jQuery ID selector	Original Code `$("#test");`
		Improved Code `$(document.getElementById("test"));`

2	Iterate an array using `for in` vs C `for` statements	Original Code
		`var array = [1,2,3 ...];`
		`for (var i in array) {`
		  `array[i] +=1;`
		`}`

		Improved Code
		`var array = [1,2,3 ...];`
		`for (var i=0; i<array.length; i++) {`
		  `array[i] +=1;`
		`}`

3	Round an integer using parseInt vs bitwise operator	Original Code
		`var number = Math.random() * 1000;`
		`parseInt(number);`

		Improved Code
		`var number = Math.random() * 1000;`
		`number | 0;`

4	querySelectorAll vs getElementsByClassName	Original Code `var items = document.querySelectorAll(".test");`
		Improved Code `var item = document.getElementsByClassName("test");`
5	querySelectorAll vs getElementsByTagName	Original Code `var items = document.querySelectorAll("test");`
		Improved Code `var items = document.getElementsByTagName("test");`

Table 1. JavaScript patterns in their original and improved form.

In order to measure the differences between running times, we created and ran simple, synthetic programs for each pattern. The observed performance improvements for the five patterns are shown in Table 2.

Trying to avoid result bias due the hardware and runtime implementation, each test was executed on several browsers on different computers. We used two desktop browsers and two Android browsers. The hardware used to run on desktop browsers was a notebook Lenovo T430, 4GB RAM, processor Intel Core i5-3320M 2.60Hz, with Windows 8 operating system. To execute on Android browsers we used a Tablet Asus MeMO Pad 7 with an Intel Atom Z3560 1.83GHz quad core processor, 2GB RAM and Android 4.4.2.

Pattern #	**PC Chrome v36**	**PC Mozilla v30**	**Android Tablet Native Browser**	**Android Tablet Chrome v36**
1	**2.15x**	1.49x	1.98x	1.79x
2	55.54x	**167.60x**	27.96x	25.26x
3	12.91x	**33.30x**	11.24x	4.75x
4	138.88x	364.97x	**915.23x**	394.26x
5	236.16x	393.29x	213.69x	146.89x

Table 2. Improvement rate for each pattern comparing slow and fast code.

As it can be seen in Table 2, the pattern #1 (using a Native Selector by ID vs a jQuery ID selector) almost doubles the performance of the code, without regard of the browser or hardware used.

Pattern #2 (using a `for-in` statement vs a standard C-style `for`) has an improvement of 20x on the Android Tablet, but it shows greater differences yet on the Mozilla browser running on the PC: a 167x improvement over the original code.

The performance improvements on pattern #3 (using parseInt vs a bitwise operator to round an integer) are more regular: from 4.75x on Android-Chrome to 33.3x on PC-Mozilla.

Pattern #4 (querySelectorAll vs getElementsByClassName) shows both, an impressive improvement on performance on all tests and a peak of 915.23x on the Android Tablet with a native browser.

Finally, pattern #5 (querySelectorAll vs getElementsByTagName) shows more regular, although high improvement values. The running times are between 146x and 393x.

It is noteworthy than the improvement in performance between the slower and the fastest version of a pattern ranges between 1.49 times for the worst case (pattern 1 on a desktop PC with a Mozilla browser) and up to 915.23 times for the best case (pattern 4 on an Android tablet). A careful selection of the patterns to replace can have great impact on the performance of the JavaScript code.

Moreover, there is no test setting showing the best improvement in all cases. Table 2 shows in boldface the best improvement for each pattern. The only test setting that was not the best in any of the patterns was the Android Table with the Chrome browser.

2.1 Solving Semantic Differences

Notice, however, that patterns 4 and 5 are optimistic transformations, since the original and improved code will generate the same result most of the time, but not always. This discrepancy exists because the `querySelectorAll` method returns an instance of a *NodeList* object, while `getElementsByClassName` and `getElementsByTagName` return an *HTMLCollection* object. Both objects, *NodeList* and *HTMLCollection* are similar, since both are collections containing the properties *length* and *item()*, which are used to get the number of items in the collection and to get an specific item, respectively. But because both collections use different constructors, a code that checks for the instance type of the collection can fail in the translated version.

Another consideration to take in account is that *HTMLCollection* objects are live collections, that is, when the DOM (Document Object Model) is updated the collection is updated. For example, after retrieving all nodes with class name *class1* by using the `getElementsByClassName` method, if a new node with *class1* class name is added to the DOM, the collection will be automatically updated.

Although for most cases the differences between *HTMLCollection* and *NodeList* will not change the semantic of the program, it may be the case. Fortunately, there is a way to circumvent this semantic difference between the two collection types. When rewriting calls to `querySelectorAll` into calls to `getElementsByClassName` or `getElementsByTagName`, it is possible to wrap the returned collection with a new *NodeList* collection. Figure 1 shows the method `createNodeList`, which converts an *HTMLCollection* object into a *NodeList* collection.

```
function createNodeList (elements) {
    var fragment = document.createDocumentFragment();
    for(var i; i < elements.length; i++)
        fragment.appendChild(elements[i]);
    return fragment.childNodes;
};
```

Fig. 1. Function that converts an *HTMLCollection* into a *NodeList*.

After applying these changes to patterns 4 and 5 and running the tests again, we found that some previous improvement rates were down. For example, in a PC using Chrome the execution time was down from 138.88 times faster to a more conservative 17.2 times faster. Still, there is a sizeable improvement in performance when using `getElementsByClassName` or `getElementsByTagName` methods instead of `querySelectorAll`.

3 Implementing a Meta-Programming and Rewriting Infrastructure

The experiments and tests mentioned in the previous sections were done by hand, just writing the different examples and running each program with the same dataset on different test settings. As we understand that rewriting the code, even for a more efficient version, is usually not an option on a production environment, we created a tool to automate the process.

In order to accelerate the process of rewriting the identified slow patterns, we designed a new JavaScript dialect and implemented a rewriting infrastructure. The resulting framework, named PumaScript, provides a general tool to experiment with code introspection and meta-programming applied to improving language performance.

PumaScript main feature is the support for meta-functions, a mechanism similar to the programmable macro-expansion systems available in Lisp [11], Ruby [26], and other programming languages. Like other macro systems, PumaScript meta-functions can expand caller expressions in-line. When called, a meta-function takes the decorated Abstract Syntax Tree (AST) of the arguments and returns an AST to be used as a replacement for the caller expression.

However, there are two key differences between PumaScript meta-functions and other macro systems:

1. Under certain conditions, a PumaScript meta-function can choose not to expand a certain occurrence of a caller expression, returning a *null* value instead of an AST.
2. PumaScript does not have a special *macro-expansion* phase before fully executing the program. Instead, meta-functions are live functions just like normal functions, and they can be called any time during the lifetime of the program.

PumaScript meta-functions can execute any arbitrary computation, including calling other normal functions or meta-functions. Additionally, all meta-functions have access to special *intrinsic functions* which provide access to the AST of the program for introspection and re-writing of any portion of it.

Figure 2 shows a simple example of PumaScript meta-function. This meta-function rewrites all its calls (`sum(x,y)`) into an addition expression (`x + y`).

The current high level execution flow of PumaScript is shown in Fig. 3. We use the Esprima library [14] to parse the JavaScript-like (JavaScript plus meta-functions) syntax. Afterwards, our own PumaScript runtime is used to execute the AST following the standard JavaScript semantics, plus the additional rules and semantics added by our extension language. Once the program is executed, PumaScript runtime discards the meta-functions nodes of the AST, and the resulting Decorated Syntax Tree (DST) is processed by the Escodegen library [28] in order to pretty print the program into standard JavaScript.

```
/* @meta */
function sum(a, b){
    return pumaAst( $a +  $b);
}

/* The following call to sum(5, 6),
will expand to the 5 + 6 expression */

sum(5, 6);
```

Fig. 2. A simple meta-function and its invocation.

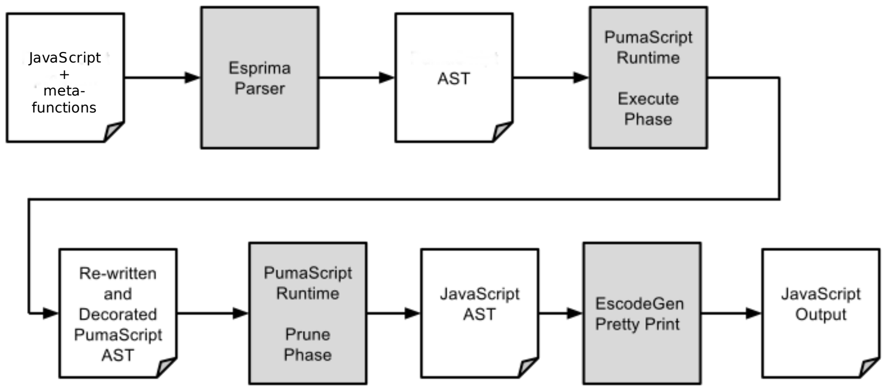

Fig. 3. PumaScript program execution workflow and high level modules.

3.1 PumaScript Meta-Functions

As seen in Fig. 2, PumaScript meta-functions are written just like normal JavaScript functions, with the annotation `@meta` in a comment before the function declaration. This method is used to avoid introducing a new syntax requirement, making our meta-programming language backward compatible with standard JavaScript.

As previously stated, meta-functions work in a similar way than regular JavaScript functions, with three specific differences:

1. All its parameters will evaluate to a reference into the caller argument DST at execution time. For example, when calling the meta-function `foo(a,b)` with actual argument expressions `(2 * x)` and `(3 * y)`, the parameter `a` will take the value of the syntax tree for `(2 * x)` and the parameter `b` will take the value of the syntax tree `(3 * y)`. Other example can be seen below (Fig. 4).
2. All meta-functions must return a valid syntax tree or *null*. If the return value is *null*, then the caller expression will not be rewritten. Otherwise, if the return value is a non-null syntax tree, the caller expression will be replaced with the returned syntax tree, actually rewriting the caller expression in the process.
3. Our meta-functions have access to a reserved context with intrinsic objects and functions. These reserved elements can be used to make program introspection or to rewrite any portion of the program. Some intrinsic objects and functions available in the meta-function context are: `pumaAst`, `context`, and `pumaFindByType`.

The example of Fig. 4 shows a meta-function that can return *null*. The meta-function `firstLetter` rewrites its caller only if the actual argument is a string literal. As it can be seen in the figure, in the second call the function returns *null* and then the caller is not rewritten.

As an example of use of intrinsic objects and functions, Fig. 5 shows the meta-function `countForStatements`. This meta-function counts all the occurrences of `for` statements in a script and shows the resulting number by using the standard console object. In this example we use the intrinsic function `pumaFindByType` and the intrinsic object *pumaProgram*. These functions and objects can be used to introspect any portion of the program, not only the current context that is calling the meta-function.

3.2 Rewriting Code

In this section we describe how automatic rewriting of the patterns presented in Table 1 was implemented using the PumaScript framework. For each pattern we had to write a meta-function that replaces the slow code by the faster code, taking in account the context where the code is used. For clarity and space reasons, we only explain in this paper the meta-functions developed for the first two patterns shown in the aforementioned Table.

```
// Program sent to PumaScript

/* @meta */
function firstLetter(valueExp){
    var ast = null;
    if(valueExp.type === "Literal"){
        ast = valueExp;
        ast.value = ast.value.substring(0, 1);
    }
    return ast;
}

// This call will be rewritten to "H";
firstLetter("Hello World");

// This call will not be rewritten because
// the argument expression is not a literal
firstLetter("Hello " + "World");

// Output of PumaScript
"H";
firstLetter("Hello " + "World");
```

Fig. 4. A meta-function replacing string literals and its invocations.

```
/* @meta */
function countForStatements() {
  var forStas = pumaFindByType(pumaProgram, "ForStatement");
  console.log("For statements found: " + forStas.length);
  return null;
}
```

Fig. 5. Meta-function counting the number of `for` statements in a program.

Pattern 1 - Rewrite jQuery Selector Calls: Figure 6 shows a meta-function that rewrites jQuery selectors by Id into the more efficient JavaScript native API `document.getElementById`. Line 6 checks that the actual argument is a *Literal* node and tests for the regular expression used to match *selectors by Id*. By executing the `then` branch, it removes the # character from the beginning and returns a new abstract syntax tree (AST) using `document.getElementById` and the provided argument with the modified string literal. The intrinsic function `pumaAst` is used to build the returned AST, starting from a template where the local variables are expanded with their actual values.

```
/* @meta */
function $(valueExp){
  var regex = /^#\b[a-zA-Z0-9_]+\b$/;
  var argValue = {};
  var substr = '';

  if(valueExp.type === "Literal" && regex.test(valueExp.value)){
    valueExp.value = valueExp.value.substring(1);
    return pumaAst($(document.getElementById($valueExp)));
  }
  else if(OPTIMISTIC_REWRITE){
    argValue = evalPumaAst(valueExp).value;
    if(regex.test(argValue)){
      console.log("WARNING: Optimistic rewrite at line("
      + valueExp.loc.start.line + ")");
      substr = valueExp.substring(1);
      return pumaAst($(document.getElementById($substr)));
    }
  }
  return null;
}
```

Fig. 6. Meta-function to rewrite jQuery selectors by Id.

The simplest use case happens when the meta-function is called with a simple string literal argument.

```
// Invocation with literal
var myElement = $("#Element_Id_1");
```

In this case, it is always safe to rewrite the invocation. Therefore, the execution of the meta-function of Fig. 6 will follow the `then` branch of the `if-else` statement of Line 6.

On the other hand, when the actual argument is not a simple literal, the meta-function uses the intrinsic function `evalPumaAst` to evaluate any portion of AST in the current execution context. In this case, it uses a flag variable `OPTIMISTIC_REWRITE` to enable optimistic rewriting when it can check that at least one execution of the caller expression matches the *selector by Id* form.

For example, the following invocation may not be safe to rewrite if the variable `element_id` is an argument into a function. Thus, by using the `evalPumaAst` intrinsic function in Line 11 of Fig. 6, the meta-function can detect that the call is safe to rewrite into a more efficient API call.

```
// Invocation with non-trivial expression
var element_id = "5";
var myOtherElement = $("#Element_Id_" + element_id);
```

Finally, there are cases where the selector does not match a simple *selector by Id*, as in the following example.

```
// Invocation that does not match a selector by Id
var myOtherClassElements = $(".Class_Id_" + element_id);
```

For this invocation, the meta-function is capable of identifying the problem easily: no matter what value the variable `element_id` takes, it will not form a valid *selector by Id*.

Pattern 2 - Rewrite `for-in` Statements: In order to implement the automatic rewriting of `for-in` statements into C-style `for` statements, a different approach is needed. It is not possible to use a meta-function like a macro call which rewrites the caller expression. Instead, the meta-function to rewrite `for-in` statements needs to use intrinsic functions provided by the PumaScript runtime environment to analyse the AST of the running program while detecting and replacing each `for-in` sentence in the program.

Figure 7 shows the main meta-function used to rewrite `for-in` statements. In line 3, it uses the intrinsic function `pumaFindByType` and the intrinsic object `pumaProgram` to match all AST nodes whose type is *ForInStatement*. Then, the function iterates on this list and uses a helper function to rewrite each `for-in` subtree in the program.

```
/* @meta */
function rewriteForIn() {
  var forIns = pumaFindByType(pumaProgram, "ForInStatement");
  console.log("For In statements found: " + forIns.length);

  for(var i = 0; i < forIns.length; i++) {
    rewriteSingleForIn(forIns[i]);
  }
  return null;
}
```

Fig. 7. Main meta-function to rewrite all `for-in` statements.

The helper function to rewrite a single `for-in` statement, shown in Fig. 8. This function has four main steps:

1. It detects if the `for-in` statement uses either a variable declaration or an existing variable as the iteration reference.
2. It creates a new AST for an equivalent C-style `for` statement.
3. It creates an `if` statement that will be used as type guard to fallback into the original `for-in` if the type of the collection expression to be iterated is not an array.

4. It replaces the original `for-in` AST node with the generated `if` AST node.

Note that the function in Fig. 8 is not marked as a meta-function, as our language extension can arbitrarily mix meta-functions with normal JavaScript functions.

```
function rewriteSingleForIn(forInAst){
  var left = forInAst.left;
  var right = forInAst.right;
  var itemName;
  var tempId;

  // Detect which kind of iteration variable it uses
  if (left.type === "Identifier") { itemName = left;}
  else if (left.type === "VariableDeclaration") {
    tempId = left.declarations[0].id;
    itemName = pumaAst( $tempId );}
  else {return;}

  // Prepare fallback version and optimized AST
  var cloneForIn = pumaCloneAst(forInAst);
  var optimizedFor = pumaCloneAst(forInAst);

  optimizedFor.type = "ForStatement";
  optimizedFor.init = left;
  optimizedFor.test = pumaAst($itemName < $right.length);
  optimizedFor.update = pumaAst($itemName = $itemName + 1);

  // Create type-guard to test runtime type
  var temp = pumaAst(function(){
    if (Array.isArray($right)) $optimizedFor;
    else $cloneForIn;});

  var tempIf = pumaFindByType( temp, "IfStatement")[0];

  // Replace original node with type-guarded one
  forInAst.type = tempIf.type;
  forInAst.test = tempIf.test;
  forInAst.consequent = tempIf.consequent;
  forInAst.alternate = tempIf.alternate;
}
```

Fig. 8. Helper function to rewrite a `for-in` statement.

To apply this rewriting meta-function, Fig. 9 shows an example of a client program that calls the meta-function to rewrite all the program's `for-in` statements. The output obtained from running the program is shown in Fig. 10.

```
var array = [1,2,3,4,5,6,7,8,9,0];
var i = 0;

// Test for-in with existing iteration variable
for(i in array){ array[i] += 1;}

// Test for-in with new iteration variable
for(var j in array){ array[j] += 1;}

// Call to meta-function to optimize for-in
rewriteForIn();
```

Fig. 9. Client program which uses the *rewriteForIn* meta-function.

```
var array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0];
var i = 0;

if (Array.isArray(array))
  for (i; i < array.length; i = i + 1) { array[i] += 1;}
else
  for (i in array) { array[i] += 1;}

if (Array.isArray(array))
  for (var j; j < array.length; j = j + 1) {
    array[j] += 1; }
else
  for (var j in array) { array[j] += 1;}
```

Fig. 10. Output generated by PumaScript after running the program of Fig. 9.

4 Applying PumaScript to Security Issues

In this section we show how our approach cannot only used to improve the performance of JavaScript programs, but also the system security. In the most direct case, we detect insecure patterns and replace the dangerous code by using PumaScript meta-functions. But we went a step ahead and modified our PumaScript framework in order to deal with the security issues associated to dependency injections.

4.1 Detection of Server-Side Insecure Patterns

Since our PumaScript framework allow us to detect certain patterns in the code and rewrite the program with other piece of code, we can use it to replace known insecure code by other, more secure, semantically equivalent code.

As an example we applied our approach to validate server-side (back-end) code written in Node.js [8]. This language was selected because its runtime was implemented by using Google's JavaScript V8 Engine [17]. We detected two known insecure code patterns: `setInterval(String, Int)` [20] function and the lack of use of *use strict* mode.

Replacing `setInterval()` Function The `setInterval` function is insecure because it uses the function `eval()`. This function allows user input to be evaluated unchecked. If the input is malicious JavaScript code, the code is executed and the system can be compromised. By using the `alertSetIntervalVulnerability` meta-function defined in Fig. 11, PumaScript rewrites the code to show a security warning in the execution console every time an insecure `setInterval` function is executed.

```
/**
 * @meta
 */
function alertSetIntervalVulnerability(){
  var newFunctionPuma = puma.pumaFindByType(pumaProgram, "ExpressionStatement");
  for (var index = 0; index < newFunctionPuma.length; index++){
    findSetIntervalVulnerability(newFunctionPuma[index]);
  }
  return null;
}
function findSetIntervalVulnerability(setIntervalAst){
  var expression = setIntervalAst.expression;
  if (expression.type === 'CallExpression'){
    var name = expression.callee.name;
    if (name === 'setInterval'){
      var parameter = expression.arguments*;
      if (parameter.type === 'Literal'){
        console.log('The defined setInterval function is a potential security problem.');
      }
    }
  }
}
```

Fig. 11. Meta-function to detect the use of `setInterval()` function.

Strict Mode Case When developing with Node.js, it is important to use the *strict mode* to opt for a restricted version of JavaScript. This version allows detection of code errors and also increase the performance of the program [21]. As shown in Fig. 12, we can use PumaScript to validate that JavaScript is being edited under the *strict mode*. Again, the result of PumaScript detection will be shown in the console as a warning.

4.2 Dependency Injector

Nowadays, most of new web based solutions are created by integrating libraries published by third parties for different purposes [24]. However, this practice is not without risks. In fact, the biggest and most popular publisher of JavaScript third party libraries, the node-package-module (npm) [22], does not ensure any security level as it does not perform safety checks on any of the packages that it publishes. This is a huge problem for the security and integrity of the systems that use this package manager. Making things worse, most key libraries do not provide a really high security level, since they do not perform any safety check [31].

```
/**
 * @meta
 */
function alertStrictModeNotFound(){
  var flag = false;
  var newFunctionPuma = puma.pumaFindByType(pumaProgram, "ExpressionStatement");
  for (var i = 0; i < newFunctionPuma.length; i++){
    if (newFunctionPuma[i].expression.raw === "use strict"){
      flag = true;
      break;
    }
  }
  if (flag === false){
    console.log("Strict mode not activated")
  }
}
```

Fig. 12. Meta-function to determine if strict mode is being used.

The combination of the JavaScript ecosystem growth and the missed security checks presents an open door for successful cyber-attacks. For example, during the attack on the Content Management Systems (CMS) platforms, an estimated of 4.5 millions of WordPress and Joomla users were hacked by using jQuery, the most popular library in JavaScript, to inject malicious code in their websites [1].

Today, it is a common practice for web solutions to use Content Delivery Networks (CDN). Their purpose is to provide web-pages or web content, such as JavaScript libraries to end-users with high speed and availability, by applying caching techniques.

Despite the fact that dependencies are a big part of the developed software, there is no a clear unique solution to prevent attacks on CDNs [15] or to avoid including a library containing malicious code. This problem is very common and affects the integrity and security of systems. It even can be used to produce a Distributed Denial of Service (DDoS) attack [23] on external systems by using reflection and amplification techniques [7]. Several dependency injectors, such as RequireJS [27], help with the task of modularizing and solving dependencies, but they do not apply security checks for third party dependencies.

To make thing worse, outdated dependencies, which leave systems vulnerable to exploits by hackers, are a very common problem [18]. This problem has been exacerbated in the last ten years due the proliferation of sources of software dependencies and the inability to make automated upgrades when security problems are discovered. Thus, developers have to keep up to date with the latest news about security issues and continually upgrade the security checks in their software. A solution to this problem would be a tool providing automated checks and upgrades.

We tackle the problem of the lack of dependency injectors with built-in security checks for third party dependencies. In particular, we can provide a way to automatically include and perform security checks for JavaScript libraries used in web solutions. To do so, we extended our PumaScript framework to include meta-functions as checks for third-party libraries' integrity, avoiding malicious code injection.

In order to use PumaScript to detect security issues described in the previous section, we developed some models and iterated on them in order to discover the better approach. The first iteration included as parameters the list of dependencies and a fixed CDN. Thus, PumaScript meta-functions were hardcoded in the runtime and the user was not able of changing the configuration. The second iteration included the parametrization of the CDN, while the meta-functions were taken from a meta-function database expressed in a JSON format file. The third and final iteration added integrity checks for external resources as described in the 2016's recommendation on Sub-Resource Integrity (SRI) [30].

The system includes the capability of getting the integrity checks from a PumaScript database and injecting the dependency only if the security and integrity are confirmed. The security features that PumaScript must test is passed as a list of meta-functions that the PumaScript injector will look for in the meta-function database. Thus, it will be able of automatically add them to the application to test.

5 Related Work

To our knowledge, source-to-source code rewriting techniques were never before used to improve JavaScript performance. However, other approaches have been developed previously to improve the performance of JavaScript programs. In this section we review other source code rewriting tools, describe commonly used JavaScript pre-processing tools, and analyze the most relevant approaches to compare them with our proposal.

5.1 Rewriting and Pre-processing Tools

Source Code Rewriting Frameworks: A number of existing frameworks and tool chains can be used to implement end-to-end source code cross-compilation or refactoring. Stratego XT [3] and DMS [2] are two of the more mature frameworks to build source-to-source transformation tools applying rewriting techniques. The main strength of these frameworks is their flexibility, as they provide end-to-end tools to build parsers, rewriting scripts, semantic analysers, and pretty printers. Our approach, as implemented by PumaScript, is simpler and does not aim to be a generic tool to transform from any source to any possible target. It is focused in JavaScript language rewriting.

As a disadvantage, to build an end-to-end JavaScript-to-JavaScript tool with these frameworks requires an important amount of work, because every major component must be created using only these tools. In contrast, for the implementation of PumaScript we reuse existing and tested components, such as the Esprima parser [14] for the front-end and EscodeGen [28] for pretty printing. Our toolchain can be easily modified by changing those components for others that share the same input/output interfaces.

Another major difference between these frameworks and our solution, is that to implement transformations in these frameworks the developer must learn specific languages based on the tree rewriting paradigm. This programming paradigm is not well known by most developers. Instead, PumaScript meta-functions use the same JavaScript language known by any JavaScript developer. Our approach does not introduce a significantly new programming paradigm beyond requiring familiarity with macro-expansion systems, which are already available in a number of well known programming languages [11, 26].

Code Minifiers and Pre-Compilers: Code minifiers have been utilized by the JavaScript community for a long time. Simple minifiers like JSMin [6] and JSZap [4] provide mostly bandwidth optimization but not measurable performance improvements.

Elaborated pre-compilers, like Google Closure Compiler [16], are capable of more advanced optimizations like code inlining and removing unused variables. However, those optimizations are provided as a mean to shorten the source code and not as a way to improve performance.

5.2 JavaScript Performance Improvement Approaches

Runtimes and Just in Time Compilers: The greatest performance improvement in JavaScript language has been related with the progression from using runtimes, like Safari SquirrelFish [10], to more advanced Just In Time (JIT) compilers, such as Chrome V8 [17], Mozilla SpiderMonkey [9], or Microsoft Chakra [19].

All of these JIT-based engines reuse techniques previously used in other language runtimes, most notably the Java HotSpot compiler [25]. Although the introduction of JIT compilers has provided an improvement of at least one order of magnitude, even the more advanced JIT engines cannot optimize certain language patterns, such as `for-in` vs C style `for`. Moreover, JIT engines are not good candidates to incorporate optimizations related to similar APIs with different performance, like the jQuery selectors vs native APIs cases identified in this work.

Language Extensions: Language extensions like Mozilla asm.js [13] or Intel proposed SIMD [5], and parallel execution [12] extensions are capable of providing important performance benefits for certain use cases. However, these approaches suffer from several limitations.

First, developers must embrace the language extensions by using them in their source code. Second, runtime providers must implement support for these extensions in their products. These limitations generate the classic chicken-and-egg problem: developers are not willing to invest effort in modifying their code until most runtime providers add support for a certain language extension, while at the same time, runtime providers are not encouraged to support the extensions because there are not enough projects using them.

In contrast, our approach can be used from day one by developers, not requiring them to change the source code and having immediate benefits in existing runtimes. The only additional cost for a developer to use our method is to add our framework (that is, PumaScript module and the rewriting scripts) to the deployment process.

6 Summary and Future Work

In this work we shown that the performance of JavaScript programs is highly sensitive to the use of a number of source code and API patterns. Therefore, sizable performance improvements can be achieved by replacing these patterns by semantically equivalent faster code.

Also, we introduce a JavaScript language extension and framework called PumaScript, which can be used to automate a number of source code rewriting tasks. This new framework adds meta-functions to JavaScript, allowing introspection and rewriting syntax trees on the fly, without requiring the program to be restarted or a specific macro-expansion phase included in the runtime.

Developers can integrate PumaScript rewriting infrastructure and transformation scripts into their continuous integration environments to optimize the source code before the deploying phase. Additionally, our novel approach to improve performance is complementary to progress in JavaScript runtimes, high performance language subsets, and other efforts to improve the language performance.

Still, there is additional work to be done in order to validate how the exhibited performance benefits in our simple experiments translate to real life code. The first open task is to analyse how common these non-optimal patterns are in actual, already in production JavaScript source code.

Second, we have to use real life code from third parties to prove that the performance gain are similar to the shown by the synthetic examples we used in this work. We plan to use open source projects from public repositories to create a new, more comprehensive set of tests.

Moreover, as it is possible that several other non-optimal patterns exist and can benefit from our approach, we plan to analyze the aforementioned open source code to discover such patterns. For this task we now have the advantage of having a functioning rewriting framework that will provide us with an environment to quickly develop experiments.

Finally, we would like to explore the use of our PumaScript infrastructure in other applications related to JavaScript meta-programming. For example, it could be applied to source code generation, construction of static, pre-compiled domain-specific languages, static and real-time source code analysis, application of aspect oriented programming, source code instrumentation, and source-to-source transformation to other scripting languages.

References

1. AVAST: Wordpress and Jommla hacked by facked libraries. `https://blog.avast.com/wordpress-and-joomla-users-get-hacked-be-aware-of-fake-jquery`
2. Baxter, I.D., Pidgeon, C., Mehlich, M.: DMS: Program transformations for practical scalable software evolution. In: ICSE 04: Proceedings of the 26th International Conference on Software Engineering. pp. 625–634. IEEE Computer Society, Washington, DC (2004)
3. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. a language and toolset for program transformation. Science of Computer Programming 72(1–2), 52–70 (June 2008)
4. Burtscher, M., Livshits, B., Zorn, B.G., Sinha, G.: JSZap: compressing JavaScript code. In: Proceedings of the 2010 USENIX Conference on Web application development. pp. 4–4. Boston, MA (June 2010)
5. Corporation, I.: SIMD in JavaScript. `https://01.org/node/1495`
6. Crockford, D.: JSMin: The JavaScript minifier. `http://www.crockford.com/javascript/jsmin.html`
7. Fastly: DDoS definitions. `https://www.fastly.com/sites/default/files/Fastly-Bizety%20DDoS%20White%20Paper.pdf`
8. Foundation, N.: About node.js. `https://nodejs.org/en/about/` (2016)
9. Gal, A., Eich, B., Shaver, M., Anderson, D., Kaplan, B., Hoare, G., Mandelin, D., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E., Reitmaier, R., Haghighat, M.R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: Programming Language Design and Implementation (PLDI 2009). Dublin, Ireland (June 2009)
10. Garen, G.: Announcing SquirrelFish. `https://www.webkit.org/blog/189/announcing-squirrelfish/` (June 2008)
11. Graham, P.: On Lisp: Advanced Techniques for Common Lisp. McGraw Hill (1993)
12. Herhut, S., Hudson, R.L., Shpeisman, T., Sreeram, J.: River trail: a path to parallelism in JavaScript. In: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications. Indianapolis, IN (October 2013)
13. Herman, D., Wagner, L., Zakai, A.: asm.js specification. `http://asmjs.org/spec/latest/` (2013)
14. Hidayat, A.: ECMAScript parsing infrastructure for multipurpose analysis. `http://esprima.org/`
15. IETF: RFC CDN. `https://tools.ietf.org/html/rfc6770`
16. Inc., G.: Google closure compiler. `https://developers.google.com/closure/compiler`
17. Inc., G.: V8 JavaScript virtual machine. `https://github.com/v8/v8/wiki` (2017)
18. Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., Kirda, E.: Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. `http://www.ccs.neu.edu/home/arshad/publications/ndss2017jslibs.pdf`
19. Miadowicz, A.: Advances in JavaScript performance in IE10 and Windows 8. `http://blogs.msdn.com/b/ie/archive/2012/06/13/advances-in-javascript-performance-in-ie10-and-windows-8.aspx`
20. Nemeth, G.: Node.js security tips. `https://blog.risingstack.com/node-js-security-tips/?utm_source=Codeship` (2015)

21. Network, M.D., individual contributors: Strict mode. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode` (2017)
22. NPM: NPM introduction. `https://docs.npmjs.com/getting-started/what-is-npm`
23. OWASP: Denial of Service definition. `https://www.owasp.org/index.php/Denial_of_Service`
24. OWASP: Third Party management. `https://www.owasp.org/index.php/3rd_Party_Javascript_Management_Cheat_Sheet`
25. Paleczny, M., Vick, C., Click, C.: The java hotspotTM server compiler. In: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium. pp. 1–1. Monterey, CA (April 2001)
26. Perrotta, P.: Metaprogramming Ruby. Pragmatic Bookshelf (2010)
27. RequireJS: Requirejs. `http://requirejs.org/`
28. Suzuki, Y.: ECMAScript code generator EscodeGen. `https://github.com/estools/escodegen`
29. Team, P.: PumaScript. `https://github.com/pumascript/puma`
30. W3C: Sub-resource integrity best practices. `https://www.w3.org/TR/SRI/`
31. ZDNET: JS hackers playground. `http://www.zdnet.com/article/an-insecure-mess-how-flawed-javascript-is-turning-web-into-a-hackers-playground/`