A Fuzzy Safari

Adventures in JavaScriptCore Vulnerability Research

J. Angra

Security Innovation

October 27, 2021

Abstract. Modern browser exploitation has shifted the focus of bug hunting to the browser's JavaScript engine. This component often trends towards high cyclomatic complexity and provides the greatest level of flexibility to a vulnerability researcher. This has lead to a variety of bugs in varying subcomponents; from compiler-, interpreter-, and optimization-based vulnerabilities to those triggered from garbage collection, WebAssembly, or the DOM.We present here, an approach to vulnerability research in modern JavaScript engines. We will primarily focus on *Safari*'s internal JavaScript engine, JavaScriptCore. A strong focus on bug discovery, crash identification, and logical reasoning provides success and failure over a variety of research approaches, both theoretical and pragmatic. We then outline our interpretation on the ideal approach to future work in this relam.

KEYWORDS: JAVASCRIPT, FUZZING, INTERMEDIATE REPRESENTATION, SPECULATIVE REASONING

1 Background

1.1 Browsers

In the thirty years since *WorldWideWeb*[2], the first web browser, the architecture of the WORLD WIDE WEB (WEB), and thus web browsers, has exponentionally grown in both complexity and use. Global strives in consumer technology have allowed billions[7] of users to access the WEB, often through a web browser. Naturally static HTML pages were insufficient; users wanted *interactive* pages, custom page styling, support for additional media types, and interconnectivity between two or more WEB resources. Today, the WEB still uses HTML, but it's architectural tours de force is the JavaScript language[i].
Modern browsers are built on layed components, running at least two processes: (1) A primary (main) browser process, for the application itself, and (2) one or more renderer process. An example architectual diagram is provided as *Figure 1*.

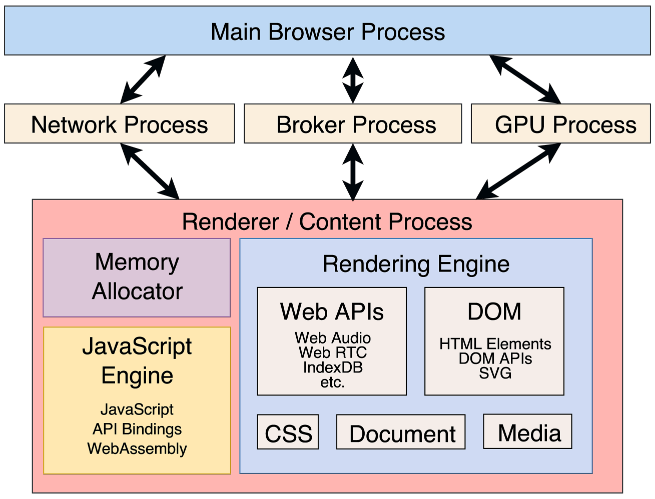

Figure 1: Process achitecture of a modern web browser.

MAIN BROWSER PROCESS This process is the primary controller for all computaiton; it displays the user interaface and manages the all processes. Safari, specifically, utilizes a split-process model[?], where each tab's web content lives in a seperate, isolated, renderer (web) process, spawned per tab. Although this model improves stability and performance, it's greatest significance is this model also provides the basis for a sandboxing infrastructure; ideally one that will limit damage to the system, if a rederer process is compromised.

[i]Amoung numerous others, e.g.: PHP, WASM, Go-lang, CSS, etc.

Renderer Process. This component parses HTML, CSS, JS, numerous image/video formats, and any other varing data formats required to construct the document. In additional to the core language features, numerous browser APIs allow developers to interact with this document, or rather Document Object Model (DOM), retreive data over the network, access web databases, and more – all via JavaScript.

From a security standpoint, the renderer retreives, lexes, parses, compiles [to C++], and displays a multitude of untrusted and potentially complex JavaScript inputs; it is the most exposed browser component. As such, it's commonly sandboxed[ii], limiting access to sensitive data and applications, both internally and within the device's operating system or underlying kernel.

1.2 JavaScript Engines

JavaScript, a weakly-typed scripting language, is one of the largest components of modern websites. Naturally, the need for high performance script exectuion resulted in deeply complex engine implementations; the perfect place to hunt for complex, domain-specific vulnerabilities. Modern JavaScript engine typically consist of a runtime and the following components:

Parser and Bytecode Compiler. The parser and bytecode compiler are responsible for converting JavaScript source code to an engine-specific bytecode representation, for use by the interpreter and JIT compiler. Parsing tokenizes the input stream and contructs an Abstract Syntax Tree (AST) based on grammartical sytax of JavaScript. The AST is subsequently compiled to bytecode.

```
// add.js
function add(a,b) {
    return a + b;
}
add(2, 1337);
```

Listing 1: Example function to JIT compiler

The bytecode is the source of truth throughout the whole engine. While some engines, such as Spidermonkey, use a stack-based virtual machine, other engines, e.g., JavaScriptCore, use a register-based virtual machine. As such, their bytecode format is fundamentally different. However, one typically common shared property is that bytecode is untyped: the bytecode operates on weakly-typed values which contain both type- and value-information. During execution, an interpreter will perform different actions depending on the runtime type of the operands.

```
// jsc -d add.js
bb#1
[   0] enter
[   1] get_scope       dst:loc4
[   3] mov             dst:loc5, src:loc4
[   6] check_traps
[   7] add             dst:loc6, lhs:arg1, rhs:arg2, operandTypes:OperandTypes(126, 126)
[  13] ret             value:loc6
```

Listing 2: Interpreter bytecode and type profile

Another property common to the resulting bytecode is that it is usually unoptimized. This is primarily done for two reason: (1) To keep the overall startup time as low as possible, forbidding the use of costly optimizations, and (2) many optimizations require type information, which is not available at this point, due to the weakly-typed nature of the language itself.

Interpreter. The task of an interpreter is to consume the bytecode and execute it. As the bytecode is specific to a given engine, so id the interpreter. Interpretation happens by fetching the next bytecode instruction from the currently executing code and dispatching it to the handler for the bytecode operation.
While interpretation of bytecode is fairly slow, in part due to the unoptimized nature of the bytecode and high dispatching overhead, the initial startup time is minimal. Thus, for simple or sparsly repeated operations it's overall better to execute it directly via the interpreter; the overhead compilation would introduce simply outweighs the faster execution speed of the resulting machine code. Conversely, if code segment/block is repeatedly executed by the application it is worth optimizing that code. This is the job of a JIT compiler. As current JIT compilers usually require type hints to produce optimized machine code, it's also the job of the interpreter to collect type profiles during bytecode execution. Often this is performed by augmenting the bytecode

[ii]The complements of this paper's work can found here <link to zach's paper/stuff>

with previously seen input types for each bytecode operation. An example of bytecode after it has gone through this process in JavaScript core is provided in *Listing 3*, below.

```
op:add,
  args: {
    dst: VirtualRegister,
    lhs: VirtualRegister,
    rhs: VirtualRegister,
    operandTypes: OperandTypes,
  },
  metadata: {
    arithProfile: ArithProfile,
  }
```

Listing 3: JavScriptCore SyntaxType for OpAdd

JIT Compiler. A JIT compiler acts as an alternative to an interpreter for script execution. It, too, consumes bytecode by the parser, as well as type profiles gathered during execution by the interpreter. Having access to both, it converts the bytecode to optimized machine code which can be executed directly on the host CPU.
As JIT compilers are crucial to execution performance, they are the subject of intensive research. Many implementations and key mechanisms have been discussed, e.g., polymorphic inline caches[iii].
A JIT compiler will often, initially, convert the bytecode to another custom intermediate representation. This IR is often graph based to facilitate the various optimizations that are later performed on it. In addition, most engines currently use a static single assignment (SSA) form to further simplify code analysis and optimization; JavaScriptCore utilizes a two-tier IR consisting of Bare Bones IR and AIR.

Speculative Optimization. One essential mechanism that allows generation of performant machine code for JavaScript and dynamically typed languages, in general, is speculations: equipped with type profiles from the interpreter, the compiler *speculates* that the same types will be used in the future. It will then guard these assumptions with runtime guards: small fragments of code that perform an inexpensive type check and bailout if the check fails, in which case execution will continue in a more generic execution tier, i.e., the interpreter. These type-guards essentially convert the previously untyped code into strictly typed code, which may subsequently be optimized in a similar fashion as other strictly-typed languages, such as C++ or Java. This mechanism is shown in *Listing 4* using an imaginary bytecode and compiler IR format.

```
function add:
  v0 = LoadArgument 0
  CheckIsInteger v0

  v1 = LoadArgument 1
  CheckIsInteger v1

  v2 = IntegerAdd v0, v1
  Return v2
```

Listing 4: Compiler IR for interpreter bytecode[iv]

It is not uncommon for an engine to have multiple levels of JIT compilers. These correspond to different optimization levels: the early JIT compilers perform less optimizations, thereby produce machine code faster. The late JIT compiler stages perform more optimizations,thereby generating faster machine code. A unit of code may be recompiled by a higher level JIT compiler when its execution count reaches another threshold. JavaScriptCore, the engine inside WebKit currently features three different JIT compilers in addition to an interpreter. V8, the engine inside the Chrome browser, on the other hand, only uses one JIT compiler and one interpreter.
Here, the function to be compiled, which is shown in *Listing 1* is invoked with integers as arguments. This is captured by the interpreter in-type profiles associated with the bytecode as shown in *Listing 2*. Based on those, the JIT compiler speculates that the same types will be used in the future and guards that assumption with two type checks. Afterwards, the compiler can use fast, specific operations, such as the addition of two integers, in lieu of slow, generic ones that cover all possible scenarios of different types. This can be seen in *Listing 3*. The integer addition used in this example could well be implemented with a single machine instruction instead of the generic addition operation as defined by the language specification.

[iii]Polymorphic Inline Caching is a broadly used mechanism to speed up polymorphic operations in dynamic language interpreters and at the same time gather type information for the JIT compiler by caching results of previous executions.
[iv]Type profiles were used to emit type checks and optimiatize specialized integer addition instructions.

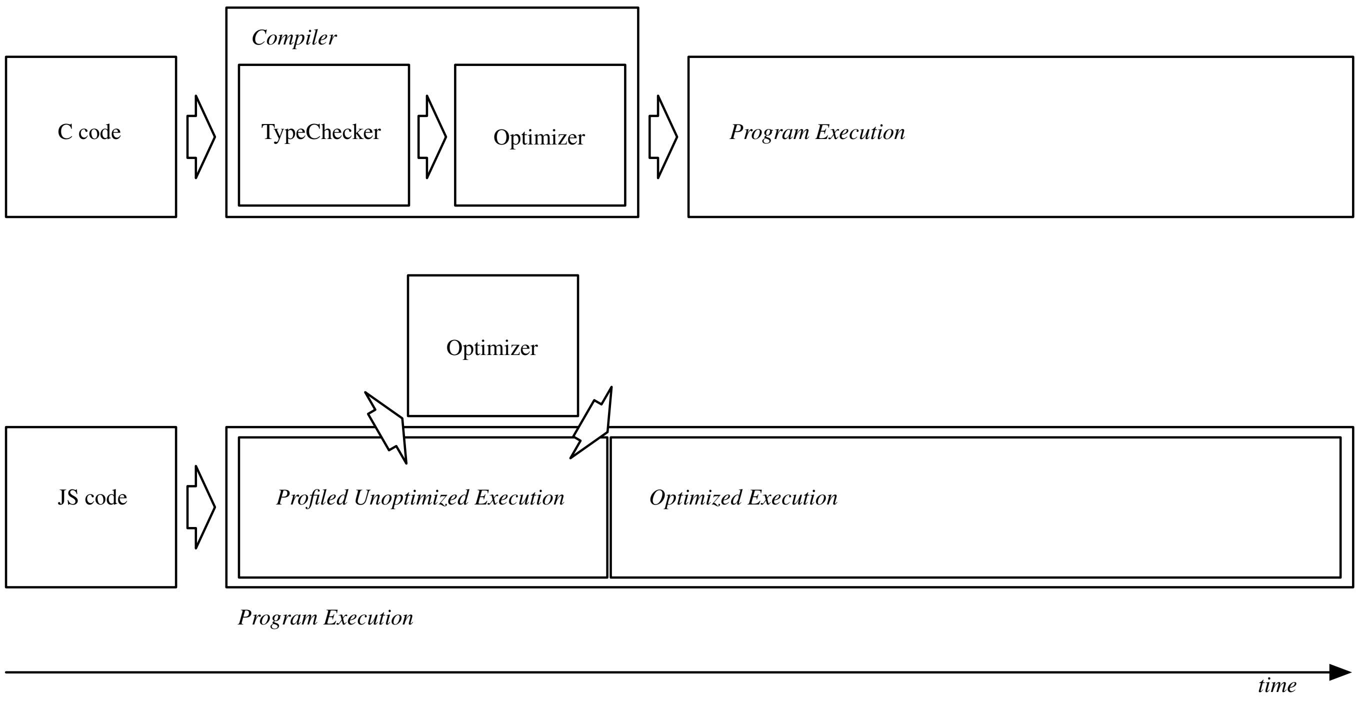

Figure 2: JSC Speculative Optimization WorkFlow

Garbage Collection. As JavaScript frees the programmer of the responsibility to return allocated memory back to the system, it requires a garbage collector (GC) to detect and reclaim unused memory.

Most high-performance garbage collector systems are based on the mark-and-sweep algorithm, which operates in two phases. First, during the periodic marking phases, the whole graph of reachable objects is scanned. Then, during the sweeping phase, every memory allocation which was not visited, implying that it is a dead object, is freed and returned to the allocator.

Various extensions of this core algorithm exist and are in use today. One key improvement lies in concurrent marking to avoid noticeable pauses in the application caused by stop-the-world garbage collection. Modern collectors such as RIPTIDE[v], the collector used in JavaScriptCore, further strive for a minimal amount of application interrupts due to garbage collection through parallelism, if multiple CPU cores are available.

[v]RIPTIDE, was not targeted for this research.

2 Motivation

2.1 State of World

The fundamental challenge of JS engine fuzzing can be summed up by the observation that, while we know the comeplte source and specification of our target source and exploitation language, respectively, the number of degrees of freedom in such an envrionment is much too large to allow for a direct, instantaneous identification of all vulnerabilities. This observation is reflected in the multitude of architectures and approaches in today's JS engine fuzzers.

The space of today's JavaScript engine fuzzers can best be understood as two mutually non-exclusive architectures: generative and mutational.

Generative fuzzers posit new test cases (i) from scratch based on a predefined grammar, e.g., DOMato[5] and Dharama[4], or (ii) by constructing them from synthesizable code blocks, e.g., CodeAlchemist[8] and Fuzzilli[6].

Mutational fuzzers posit new test cases using segments either learned from seed-based mutations or borrowed from other corpora programs, e.g., Skyfire[13], Fuzzilli[6], Nautilus[1], Superion[3], and DIE[10].

2.2 JSC Speculative Type [Confusion]

One of the more prominant observed vulnerabilty classes, type confusion vulnerabilities often require a complex control and/or data flow within the phases of an optimization pipeline. During a given optimization phase, JavaScript code is profiled, variable types are speculated, and optimized bytecode is produced[vi]. A side-by-side view of JSC's speculative optimzation workflow and a standard C-based compiler's equivalent is provided in Figure 2.

- One of the goals of DFG is to optimize away redundant operations such as type checks
- Only a few operations can alter an objects type
- If no such operation is encountered, it can be assumed that the object type will stay unchanged and a single type check will suce to prove the type of a specific value until some potentially dangerous operation is encountered
- Operations which may invoke arbitrary JavaScript are dangerous
- The arbitrary JavaScript may execute any operation, including things that may mutate an objects type
- Important to model which operations may or may not do this in order to invalidate previously proven types
- We may cause arbitrary type confusions by mutating object types

2.3 WebAssembly

- WASM growing in popularity, in general
- Research into WASM-based fuzzing of JS engines underway, but not much novel research published
- WASM code is not JIT'd, thus it utilizes a different pipeline for execution, one that will not affect out primary research

3 Methodology

Rather than build a fuzzer from re-hashed ideas, the work presented in this paper expands the work of Samuel Groß's *Fuzzilli*.

3.1 Improving Our Intrumentation

Fuzzilli's execution model critically relies on Swift's RunLoop Scheduling interface[9]. RunLoop's provide developers a programmatic interface to objects of an *input source*; for our work, input sources take the form of an event queue. Thus, the fuzzing process defaults to corpora sample execution, only diverging for *observered* events whitnessed in the RunLoop event queue.

To take complete advantage of our hardware, Fuzzilli includes an inter-machine communication module, synchronizing instances over a simple TCP-based protocol. This design allows the fuzzer to scale to many cores on a single machine as well as to many different machines. One particularly important caveat: TCP synchronization introduced a (fairly sizeable) execution bottleneck. As a result, three alternative synchronization and one alternative instrumentation models were introduced and tested:

Native IPC Unix Domain sockets with great deal of shared memory was tested first. Swift does not like IPC *unless it uses ~~Mach ports~~* ~~we were working completely in a debia~~n-based linux distro

[vi]Poor type speculation results in execution a priori; JSC defaults to unoptimized interpreter bytecode execution.

CONTAINER-ISOLATED IPC Portability, amoung effeciency, are of vital importantce; we simply isolates the fuzzing system using Docker.

Overhead of Docker was negligible relative to the aforementioned IPC architecture.

INTER-PROCESS COLLECTIVE COMMUNICATION (IPCC) Inspried by HPC execution models, the C-based OpenMPI *scatter* and *reduce* functions were used for corpora distribution and sychronization, respectively.

A visual representation of the *Scatter-Reduce-Repeat* collective communication model is provided in Figure 3. All fuzzing nodes share corpora slices with each other, deduplicating and synchronizing samples in real-time, at regular intervals.

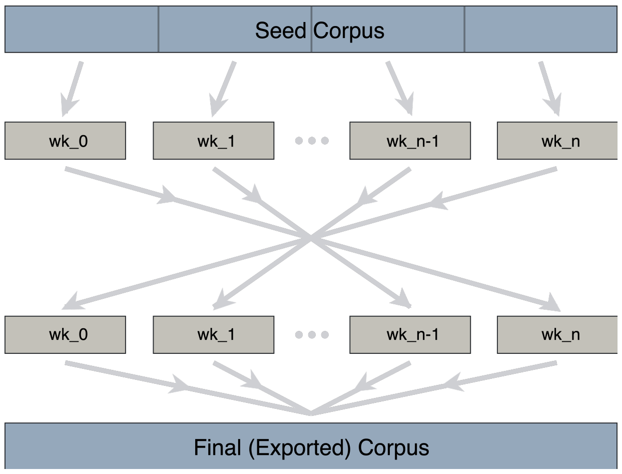

Figure 3: Scatter-Reduce-Repeat MPI fuzzing topology. Each cycle, N-equipartitions of the seed corpus are ingested by N fuzzing nodes, which operate for some time, then corpora slices concatenate and deduplicate into a single corpus.

SKIPPING THE PARSER The primary components of focus for this work are JSC's JIT compilers – do we really need to fuzzing the *entire execution pipeline*? JS engine vulnerabilities often require a complex execution flow to trigger a crash; fuzzing the entire pipeline ensures that we *can* reach all components. Furthermore, only syntatically valid samples make it past JSC's LLInt.

If we can assert the syntatic validity of our samples, LLInt could be skipped, effectively resulting into direct fuzzing of LIBJAVASCRIPTCORE. Thus, we hypothesize that such an architecture would greatly improve execution speeds and results. To test such a hypothesis Rust Foreign-Function Interfaces (FFIs) into LIBJAVASCRIPTCORE native C API were generated using bindgen.This provided direct access to fuzz JSC's type system definitions, such as *JSObject* and *JSValue*

Continued exploratory testing revealed potential in this approach and merits deeping investigative experiementation.

3.2 EXTENDING CODE GENERATION: WASM

Added wasm as an additional (minor) target $\rightarrow$ wasm isn't JIT'd so it's a completely different control flow, but is still a fresh section and had potential to uncover deep crashes

Constructors	Static Methods
_.Global	_.instantiate()
_.Module	_.instantiateStreaming()
_.Memory	_.compile()
_.Instance	_.compileStreaming()
_.Table	

Table 1: JSObject WASM contructors and static methods introduced to Fuzzilli from the WEBASSEMBLY *namespace.*

3.3 STREAMLINING TRIAGE

As exporatory testing proceeded, it became painfully obvious that crash sample automation would be required; Fuzzilli would report a crash count of five significant figures; however, not all were valid – futhermore, only a relatively infintesimal portion of crashes led to a valid bug. To futher complicate triage, fuzzing was performed on a Linux distrobution, testing against the Linux build of JSC. This build differs from the macOS flavor.

As a result, traige automation took the form of a custom script runner, executing crash samples found on the Linux fuzzing host against the *latest* ASAN build of JSC on macOS.

4 Results

4.1 Processing Speed

- An AOT transpiler (JavaScript ↔ B3IR) would allow for direct targeting
- Bypassing parseing and lexing would allow for fuzzing via C++ types
- swift compiler is awesome, openMPI is awful
- improving runloop effeciencies in Swift GCD, or rather – using concurrency-friendly language
- distrubuted dataflow over N procs > collective IPF over N procs

4.2 WASM

- → should be done independently
- takes on an approach of it's own between WASM, WAT, and JS
- we did not have the time to focus on this part for the depth required (for success)

4.3 Triage Wokflow Improvements

- Improved workflow did great!
- we implemented it, too late
- CVE–2020–9800 (overview of crash and why we didn't find it first)

5 Future Work

5.1 Lessons Learned

Thus far, we have demonstrated that effective fuzzing leverages a multitude of techniques to discover bugs, many of which require a great deal of education and practice. A recent emerging one utilizes *semantic-aware fuzzing* to drive both the generative and mutation engines. Skyfire[13] was one of the earliest research efforts to tackle the semantic problem in language fuzzing; it learns the semantics of a language from existing test cases, in the form of probabilistic context-sensitive grammar (PCSG). This grammar is then used for further fuzzing. Conversely, with a focus on stressing specific components in a JavaScript Engine, DIE[10] analyzes and utilizes the overall semantic properties of each existing test case (i.e., aspects). In this section, we propose a future (alternative) direction for continued reserach of JavaScript engines. Unsurprisingly, we, too, start by tackingly the semantic problem of language – specifically, ECMAScript.

5.2 A New Approach to the Semantic Problem

ECMA-262[12], the official language specification for which JavaScript is derived, is over 800 pages. Compared to the current ISO International C++20 standard[11] (1,800 pages), less than a thousand pages should be *more* than approachable – right? (hell no) So why does JavaScript feel so much larger of a language...

An Aside...

JavaScript is a prototype-based, weakly-typed, dynamic language. For those defined, a *type object* is a JavaScript object representing a type. Type objects define the layout, stride, and size of a continuous region of memory. There are three basic categories: *primitive*, *struct*, and *array* type objects.

Primitive type objects are type objects without any internal structure. All primitive type objects are predefined in the system.

```
var any;                      //  any (uninitialized)

var u8  = 254;                //  8-bit unsigned integers
var u16 = 65534;              // 16-bit unsigned integers
var u32 = 4294967294;         // 32-bit unsigned integers
var u64 = (2**64 - 1);        // 64-bit unsigned integers


var i8  = -127;               //  8-bit signed integer
var i16 = -32767;             // 16-bit signed integer
var i32 = -2147483647;        // 32-bit signed integer
var i64 = -2**63;             // 64-bit signed integer

var f32 = 1.123456;           // 32-bit floating point
var f64 = 1.123456789012345;  // 64-bit floating point

var str = "";                 // String primitive
var obj = Object              // Object primitive
```

Listing 5: Primitive type objects

Struct type objects can be composed as structures using the StructType constructor:

```
var Point = new StructType({ x:int8, y:int8 });
var Line  = new StructType({ from:Point, to:Point };
```

Listing 6: Struct type objects

Listing 6 constructs two new type objects called *Point* and *Line*, a structure with two 8-bit integer fields, *x* and *y*, and a structure containing two aforementioned *Point* structures. The size of each *Point* will be two (2) bytes, while each *Line* will be four (4) bytes in total; memory is laid out continuously. Hence, structures can embed other structures, e.g., *prototypes*.
Array type objects are constructed by invoking the arrayType method on the type object representing the array elements:

```
var Points  = Point.arrayType(2);
var Line2   = new StructType({ points:Points });
var Plane   = Line.arrayType(768).arrayType(1024);
```

Listing 7: Array type objects

In *Listing 7*, the type *Points* is defined as a two-element array of *Point* structures. Array types are themselves normal type objects, and hence they can be embedded in structures. The array type *Points* is then used to create the structure type *Line2*. *Line2* is equivalent, in layout, to that of the *Line* type; however, it is defined using a two-element array in lieu of two distinct fields. The *Plane* type creates a multideminsional array by invoking the constructor multiple times, resulting in a 1024*x*768 matrix of *Points*.

Reiterating our state-of-affairs: We have the entire [dynamic, weakly-typed, prototyped-based] language as our disposal; syntactic correctness difficult to determine, semantic equivalence[vii] even more so. The standard slew of problems are faced; insufficient CFG/DFG mutation controls, insufficient targeting leading to resource waste, [...]. The ECMA-262 language space is simply far too large.

To improve semantic control, an AST can be used; to improve speed, bytecode can be used; A custom intermediate representation could even be written. The aforementioned techniques aim to reduce the degrees-of-freedom of the, semantically sound and syntacticaly correct, input space. Narrowing the constraints on the input space helps to reduce complexity and improve target accuracy; we posit that we can recieve the same benifit while maintaining degrees-of-freedom.

[vii] Optimizations of this problem are exactly what JIT compilers were designed for.

5.3 JavaScript Semantics: A Breakdown

5.3.1 What is *Resolution*?

Resolution is defined here ala *Display Resolution*. In other words, as degrees-of-freedom vary, so too does our observable. In QFT, we know this term as *dimensionality* or *cardinality*; we use the former and latter interchangeably.

Classical representations of programming languages aim to reduce dimensionality in an effort to ease lexical, syntactical, and semantical analysis. This result is a *discrete resolution* of a provided input sample program, P. Examples of the aforementioned discrete resolutions can be see in *Appendix A: Cononical Representations*.

5.3.2 A Modest Proposition

Given a vector representation of a sample program, $P_{|\psi\rangle}$, renormaliztion group transformations of $|P\rangle$ will validate universality for cononical representations, P_γ, P_ρ, P_τ. As a result, dimensionality can be preserved while maintaining a discrete resolution, i.e., logically equivalent semantical mutations exist for all semantic mutations at all resolutions of P.

Challanges Assigning a semantic label to a code snippet (such as a name to a method) is an example for a class of problems that require a compact semantic descriptor of a snippet. The question is how to represent code snippets in a way that captures some semantic information, is reusable across programs, and can be used to predict properties such as a label for the snippet. This leads to three challenges:

1. Representing a snippet in a way that enables RG transformations
2. Learning which parts in the representation are relevant to prediction of the desired property
3. Learning the order of importance (precedence) of the part

5.4 Degrees of Freedom and Grammar-based Fuzzing

5.5 Chalk Trait SMT

6 Acknowledgments

We're extremely thankful to *Security Innovation* for providing the opportunity to perform research of our choosing. We'd also like to thank Joe Basirico for pushing us all to do *fail-if-you-can* research. Finally, where would we be if not for our favourite rubber duckies / advisors – Zak D., Dinesh S., and Izzy G. You ensured we fostered our curiousity, questioned our assumptions, and stayed on target – whatever that ended up being. Thank you!

7 Appendix A: Canonical Representations

7.1 Abstract Syntax Tree

A tree representation for the abstract syntactic structure of source code
→ **Node**: construct, such as statement, loop
→ **Edge**: containment relationship

This makes it possible to apply all kinds of **syntax-directed** translation/transformation ASTs are simplified parse tree. It retains syntactic structure of code.

Definition 1 (Abstract Syntax Tree). An Abstract Syntax Tree (AST) for program sample P is a tuple, where for:

$$P_{AST} = (\eta, \tau, \chi, s, \delta, \phi) \tag{1}$$

- η is a **set of nonterminal nodes**
- τ is a **set of terminal nodes**
- χ is a **set of values**
- $s \in \eta$ is the **root node**
- $\delta : \eta \to (\eta \bigcup \tau)\star$ is a function that maps a nonterminal node to a list of its children
- $\phi : \tau \to \chi$ is a function that maps a terminal node to an associated value.

Every node except the root appears exactly once in all the lists of children. Next, we define AST paths. For convenience, in the rest of this section we assume that all definitions refer to a single P_{AST} .

An AST path is a path between nodes in the AST, starting from one terminal, ending in another terminal, and passing through an intermediate nonterminal in the path which is a common ancestor of both terminals. More formally:

Definition 2 (AST path). An AST-path of length κ is a sequence of the form $n_1 d_1 \ldots n_\kappa d_\kappa n_{\kappa+1}$ where:

- $n_1, n_{\kappa+1} \in \tau$ are terminals
- $\forall i \in [2..\kappa] : n_i \in \eta$ are nonterminals
- $\forall i \in [1..\kappa] : d_i \in \{\uparrow, \downarrow\}$ are movement directions (either up or down in the tree), e.g.,

$$(d_i = \uparrow) \Rightarrow n_i \in \delta(n_{i+1}) \tag{2}$$

$$(d_i = \downarrow) \Rightarrow n_{i+1} \in \delta(n_i) \tag{3}$$

For an AST-path $\rho \in P$, we use $start(\rho)$ to denote n_1 – the starting terminal of ρ, and $end(\rho)$ to denote $n_{\kappa+1}$ – its final terminal.

Using this definition we define a path-context as a tuple of an AST path and the values associated with its terminals:

Definition 3 (AST path-context). Given an AST path ρ, its path-context is a triplet $(\chi_s, \rho, \chi_\tau)$ where $\chi_s = \phi$ (i.e., $start(\rho)$) and $\chi_\tau = \phi$ (i.e., $end(\rho)$) are the values associated with the start and end terminals of ρ, respectively.

That is, a path-context describes two actual tokens with the syntactic path between them.

Example 1. A possible path-context that represents the statement: $x = 7$;" would be:

$$(\ x,\ (NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr),\ 7\ ) \tag{4}$$

7.2 Directed Acyclic Graph

DAGs are similar to an AST but with a unique node for each value.

7.3 Control Flow Graph

A representation, using graph notation, of all paths that might be traversed through a program during its execution. A directed graph (usually for a single procedure) in which: - Each node is a single basic block - There is an edge $b1b2$, where control may flow from the last statement of $b1$ to the first statement of $b2$ in some execution.

Note: CFGs are actualy conservative approximations of the control flow

7.4 Code Property Graph

A code property graph is a property graph, $G = (V, E, ,)$, constructed from the AST, CFG, and PDG of source code with:

$$V = V_A \tag{5}$$

$$E = E_A \bigcup E_C \bigcup E_P \tag{6}$$

$$= _A \bigcup _C \bigcup _P \tag{7}$$

$$= _A \bigcup \ P \tag{8}$$

where we combine the labeling and property functions with a slight abuse of notation.

7.5 Program Dependence Graph

A directed graph representing dependencies among: - Code **Control** dependence - A's control depends on B, if Bs execution decides whether or not A is executed - Code **Data** dependence (Data Dependency Graph – DDG) - A's data depends on B if A uses variable defined in B

A PDG contains both **control dependence edges** and **data dependence edges**.

7.6 Points-to Graph

For a program location, for any object ‘reference/pointer‘, calculate all the possible ‘objects/variables‘ it may/must refer/point to: - Connect together analyzed program semantics for individual methods - Essential to expand intra-procedural analysis to inter-procedural - Detect consistent usage of resources - File ‘open‘/‘close‘, ‘lock‘/‘unlock‘, ‘malloc‘/‘free‘

7.7 Property Graph

A property graph $G = (V, E, ,)$, is a directed, edge-labeled, attributed multigraph where V is a **set of nodes**, E is a **set of directed edges**, and $: E$ is an edge labeling function assigning a label from the alphabet to each edge.

Properties can be assigned to edges and nodes by the function:

$: (V \bigcup E) KS$

where K is the set of *property keys* and S the set of *property values*/

7.8 Call Graph

A directed graph representing caller-callee relationship between methods/functions $\to$ **Node**: methods/functions $\to$ **Edges**: calls

7.9 Static Single Assignment Form

A program is in *SSA* form iff: 1. **each variable** is assigned a value in **exactly one** statement 1. **each variable** use is **dominated by the definition**

The *SSA* Graph is a directed graph in which: $\to$ **Node**: All definitions and uses of *SSA* variables $\to$ **Edges**: $\{(d,\ u)\ :\ u\}$ use the *SSA* variable defined in d

7.10 Three-Address Code

A term used to describe many different representations of the form:

$$\psi \ = \ \theta_{fn} \ \ op1 \ \ op2 \ \ op3 \tag{9}$$

where $\psi \in \Psi$, $\theta \in \Theta$ are our statement and operation value, respectively $-$ $op\star$ are optional parameters passed to θ_{fn}.

7.11 Domain-Specific Languages

We define $A+B$ as the disjoint union of two sets, A and B:

$$\begin{aligned} A+B &\equiv A \ \bigcup{}^{*} \ B \quad (10)\\ &\equiv \ (\ A\{0\}\ )\ \bigcup\ (\ B\{1\}\ ) \quad (11)\\ &\equiv \ A^{*} \ \bigcup \ B^{\ *} \quad (12) \end{aligned}$$

Note: Sequences in the free monoid are denoted as a vector: $\vec{v} \in A^{*}$

7.12 Context-free Grammar

A context-free grammar is a tuple (Σ, X, R, s) where Σ and X are finite sets called **terminals** and the **non-terminals**, respectively.

- $R \subseteq X \times (X+\Sigma)^{\star}$ is a finite set of *production rules*
- $s \in X$ is called the *start symbol*

The **language** of a context-free grammar is given by:

$$L(G) = \{\ \vec{u} \in V^{\star} \mid s \to_R \vec{u}\ \} \tag{13}$$

where the rewriting relation, $(\to_R) \subseteq (V+X)^{\star} \times (V+X)^{\star}$, is traditionally defined as the transitive closure of the following directed graph:

$$\left\{ \ \ (\vec{u}\ x\ \vec{w}, \ \ u\ \vec{v}\ w) \ \ \middle| \ \ \vec{u}, \vec{w} \in (V+X)^{*}, \ \ (x, \vec{v}) \in R \ \ \right\} \tag{14}$$

Note: Given a set S, the free monoid on S is the set S^{*} of all finite sequences of elements $s \in S$, made into a monoid using concatenation.

7.13 Intermediate Representations

7.14 Assembly Instructions

7.15 Machine Code

REFERENCES

[1] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, *Nautilus: Fishing for deep bugs with grammars.*, 2019 Annual Network and Distributed System Security Symposium (NDSS), 2019.

[2] Tim Berners-Lee, *Worldwideweb, the first web client*, (1990).

[3] B. Chen, J. Wang, L. Wei, and Y. Liu, *Superion: Grammar-aware greybox fuzzing*, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 724–735.

[4] Christoph Diehl, *Dharma*, (2015).

[5] Ivan Fratric, *The great dom fuzz-off of 2017*, (2019).

[6] Samuel Groß, *Fuzzil: Coverage guided fuzzing for javascript engines*, Master's thesis, 2018.

[7] Miniwatts Marketing Group, *Internet world stats: Usafe and population statistics*, (2020).

[8] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha, *Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines.*, 2017 Annual Network and Distributed System Security Symposium (NDSS), 2019.

[9] Apple Inc., *Processes and threads*, 2021.

[10] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, *Fuzzing javascript engines with aspect-preserving mutation*, 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 1629–1642.

[11] Richard Smith, *Iso international standard iso/iec 14882:2020(e) programming language c++*, ISO/IEC, 01 2021.

[12] TC39, *Ecmascript 2021 language specification*, Ecma International, 06.

[13] J. Wang, B. Chen, L. Wei, and Y. Liu, *Skyfire: Data-driven seed generation for fuzzing*, 2017 IEEE Symposium on Security and Privacy (SP), 2017, pp. 579–594.