Towards a Verified Range Analysis for JavaScript JITs

Fraser Brown
Stanford, USA

John Renner
UC San Diego, USA

Andres Nötzli
Stanford, USA

Sorin Lerner
UC San Diego, USA

Hovav Shacham
UT Austin, USA

Deian Stefan
UC San Diego, USA

Abstract

We present VeRA, a system for verifying the *range analysis* pass in browser just-in-time (JIT) compilers. Browser developers write range analysis routines in a subset of C++, and verification developers write infrastructure to verify custom analysis properties. Then, VeRA automatically verifies the range analysis routines, which browser developers can integrate directly into the JIT. We use VeRA to translate and verify Firefox range analysis routines, and it detects a new, confirmed bug that has existed in the browser for six years.

CCS Concepts: • **Security and privacy** → **Browser security**; • **Software and its engineering** → **Just-in-time compilers**; **Software verification and validation**; **Domain specific languages**.

Keywords: JIT verification; range analysis; JavaScript

ACM Reference Format:
Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Towards a Verified Range Analysis for JavaScript JITs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3385968

1 Introduction

On May 30, 2019, employees of the cryptocurrency startup Coinbase were targeted by a phishing campaign that lured them to visit a Web page hosting attack code [88]. This code exploited previously unknown bugs in Firefox to take over the victim's machine; the first bug arose from incorrect type deduction in the *just-in-time* (JIT) compiler component of Firefox's JavaScript engine [15].

Earlier this year, Google's Threat Analysis Group identified websites, apparently aimed at people "born in a certain geographic region" and "part of a certain ethnic group," that would install a malicious spyware implant on any iPhone used to visit them. Two bugs exploited in this campaign, according to analysis by Google's Project Zero [41, 68], were in the JIT component of Safari's JavaScript engine [5, 34].

The JavaScript JITs shipped in modern browsers are mature, sophisticated systems developed by compilers experts. Yet bugs in JIT compilers have emerged in recent months as *the single largest threat to Web platform security*, and the most dangerous attack surface of Web-connected devices.

Unlike other compilers, browser JITs are exposed to adversarial program input. Remote attackers can craft JavaScript that will trigger JIT compiler bugs and break out of the sandbox on victim users' machines. These attacks are possible in spite of the fact that JavaScript is a memory-safe language, because its safety and isolation guarantees only apply if they are correctly maintained by JIT compilation.

This is easier said than done. Consider JavaScript arrays, which are maps from index positions to values. These maps can be "sparse," in the sense that indices can be missing in the middle. Any out-of-bounds accesses must be checked to return the special value `undefined`. Furthermore, values stored in an array can be of any type, so array elements must be tagged or boxed. In a naïve implementation, numerical kernel performance would be unacceptably slow.

JavaScript JITs speculatively optimize accesses to dense arrays and to arrays whose elements are all the same type, with bailouts to a slow path should the array shape change. And they perform *range analysis* on values that could be used as array indices to facilitate bounds-check elimination. When range analysis confirms that the array indices are guaranteed to be within a given range, the JIT compiler generates code that allocates the array sequentially in memory and allows sequential access without any bounds checks.

These optimizations are crucial—but also risky. Failing to bail out of the speculative fast path when an array's shape changes leads to type confusion; incorrectly eliminating a bounds check allows out-of-bounds memory accesses. Both bug types can be exploited for arbitrary code execution [63].

The implications of JavaScript JIT bugs for security were recognized as early as 2011 [103]; attackers have turned to JIT bugs as other avenues for browser exploitation become more rare and more difficult to exploit (see, e.g., [121]). Since late

2017, industrial security researchers have uncovered a dozen or more bugs in the JIT compilers shipping with Chrome (e.g., [19, 22, 23]), Firefox (e.g., [3, 15, 18, 20]), Safari (e.g., [5, 16, 17, 21, 34]), and Edge (e.g., [12–14]). They have documented JIT compiler internals and developed generic techniques for exploiting JIT compiler bugs for code execution in blog posts [43, 59, 60, 101, 105, 110, 117, 124], in industrial security conference presentations [58, 64, 67, 77, 104, 115], and in industrial journals such as *Phrack* [69]. And, as we noted above, JIT bugs are being weaponized against real users.

In short, the status quo is untenable. The JIT's adversarial setting means even obscure bugs that no normal program would *ever* hit become exploitable attack vectors. To secure the Web, it is necessary to build and deploy JIT compilers free of security bugs.

In this paper, we explore, for the first time, the possibility of using *compiler verification* to secure browser JavaScript JIT compilers. Verification proves that the compiler is correct using formal methods that consider all possible corner cases.

There is, of course, much prior work on compiler verification (§8). But JavaScript JITs are a new and challenging domain for verification. JavaScript is an unusual language with complicated semantics; the ARMv8.3-A architecture revision even adds an instruction, `FJCVTZS`, to support floating-point conversion with JavaScript semantics [1]. And because browser JITs are supposed to improve the perceived runtime performance of Web programs, compilation time is a cost. A JIT that is verified but slow will not be acceptable.

As a first step in this direction, we build a system, VeRA, for expressing and verifying JavaScript JIT range analyses. VeRA supports a subset of C++ for expressing range analysis routines, and a Satisfiability Modulo Theories [36] (SMT)-based verification tool that proves the correctness of these analyses with respect to JavaScript semantics. Browser developers can write their range analysis code using VeRA C++, prove it correct, and incorporate the verified code directly into their browser.

Compared to prior work in compiler verification, VeRA distinguishes itself by handling the details of a realistic range analysis: we use VeRA to express and verify the range analysis used *in practice by the Firefox browser*. This requires handling many challenges: complicated corner cases of JavaScript semantics; complex dataflow facts whose semantics are often disjunctive predicates; and complex propagation rules for those dataflow facts. Our verification uncovered a new Firefox range analysis bug and confirmed an old bug from a previous version of the browser. We also find that our verified routines work correctly in Firefox—they pass all 140 thousand Firefox JIT tests—and perform comparably to the original routines in both micro and macro benchmarks.

2 Overview

This section gives an overview of range analysis in JIT compilers and the ramifications of range analysis bugs. Then, it walks through using VeRA to verify a piece of the Firefox JIT's range analysis logic.

2.1 Range Analysis in JITs

Range analysis is a dataflow analysis that compilers use to deduce the range—typically upper and lower bounds—of values at different points in program execution. These range deductions, or value constraints, are then used by different optimization passes to generate efficient code. For example, Firefox's dead code elimination (DCE) pass eliminates blocks guarded by contradictory constraints [6]:

```
if (x > 0)
  if (x < 0) /* ... dead code... */
```

It also eliminates redundant checks—in the redundant check elimination (RCE) pass—when it can prove that values are within a certain range [8]:

```
var uintArray = new Uint8Array(...); // buffer

function foo(value) {
  if (value >= 0) { // always true; redundant check
    return value;
  } else { /* ... dead code ... */ }
}

for(let i = 0; i < uintArray.length; i++) {
  foo(uintArray[idx]); // call foo with unsigned value
}
```

Here, the comparison in `foo` is always true—`uintArray` can only contain unsigned integers—and thus can be eliminated (along with the **`else`** branch).

More significantly, Firefox relies on range analysis to move and eliminate internal JavaScript array bounds checks [7]. Since JavaScript is a memory safe language, the compiler inserts bounds checks around every array access. For instance, in the example above, the array indexing operation `uintArray[idx]` internally performs a bounds check, which returns **`undefined`** if the access is out of bounds, i.e., when `idx < 0` or when `idx >= uintArray.length`. In practice, this incurs overhead—real applications make heavy use of arrays—and JITs aggressively try to eliminate bounds checks. In the example above, for instance, Firefox can prove that the array accesses are in-bounds and eliminate *all* the internal bounds checks.

2.2 From Range Analysis Bugs to Browser Exploits

Bugs in the range analysis code can, at best, manifest as application correctness errors—and at worst as memory safety vulnerabilities. For example, an incorrect range deduction can cause the JIT to eliminate a bounds check, which can in turn allow an attacker to read and write beyond JavaScript array bounds and hijack the control flow of the browser renderer process (e.g., using JIT-ROP techniques [84, 109]).

```
class Range : public TempObject {
  int32_t lower_;
  int32_t upper_;
  bool hasInt32LowerBound_;
  bool hasInt32UpperBound_;
  // possibly not a whole number
  FractionalPartFlag canHaveFractionalPart_;
  // possibly negative zero
  NegativeZeroFlag canBeNegativeZero_;
  // the maximum exponent needed to represent the number
  // 0-1023, 1024 indicates inf, 65536 indicates NaN or inf
  uint16_t exp_;
}
```

Figure 1. Parts of Firefox's range object

```
Range* Range::rsh(TempAllocator& alloc, const Range* lhs,
↪  int32_t c) {
  MOZ_ASSERT(lhs->isInt32());
  int32_t shift = c & 0x1f;
  return Range::NewInt32Range(alloc, lhs->lower() >> shift,
  lhs->upper() >> shift);
}
```

Figure 2. Firefox's range analysis logic for the right shift operator.

```
range rsh(range& lhs, int32_t c) {
    int32_t shift = c & 0x1f;
    return newInt32Range(lhs.lower >> shift,
    lhs.upper >> shift);
}
```

Figure 3. Simplified VeRA implementation of the right shift operator.

Until recently, TurboFan—the JIT compiler component of Chrome's V8 JavaScript engine—used to deduce that `indexOf`, when applied to a string, would return an integer in the range $[-1,$ `String::kMaxLength`$-1]$, where `String::kMaxLength` is the longest allowed V8 JavaScript string ($2^{28} - 16$ characters). Unfortunately, the actual V8 implementation of `indexOf` can return `String::kMaxLength`—one more than the range analysis deduced. As the following example (from the original bug report [22]) shows, this bug allowed attacker-supplied JavaScript to create a variable `i` that TurboFan deduced to be 0, but actually held an arbitrary value (in this case, 100, 000):

```
var i = 'A'.repeat(2**28 - 16).indexOf("", 2**28);
i += 16; // real value: i = 2**28, optimizer: i = 2**28-1
i >>= 28; // real value: i = 1, optimizer: i = 0
i *= 100000; // real value: i = 100000, optimizer: i = 0
```

Since TurboFan thought `i` was zero, it would eliminate all bounds checks on `i`—so attackers could use `i` as an index into *any* array in order to access that array out-of-bounds.

2.3 Why Range Analysis is Hard to Get Right

The `indexOf` example is not the only case of an exploitable browser vulnerability introduced by a buggy JavaScript range analysis. Similar bugs have been a problem in practice for all major browsers. This is because JavaScript range analysis is hard to get right.

First, it requires reasoning about double-precision floating point numbers. To do so correctly and efficiently, browsers can't just implement ranges as lower-bound and upper-bound pairs; instead, their range objects are necessarily complicated structures. Figure 1 shows Firefox's range object, which keeps track of, among other things, integer bounds, special values, and whether the range includes non-integrals. Not only is the data structure itself complex, but there are also subtle invariants that it must maintain.

Second, tracking special floating-point values like NaN, Infinity, -Infinity, and -0.0 is error-prone. For example, until recently, Turbofan's range analysis incorrectly deduced that `Math.expm1`, the JavaScript builtin used to compute $e^x - 1$, must either return a number value or NaN—it didn't account for floating point negative zero. The browser's implementation of `Math.expm1`, applied to -0.0, correctly returned -0.0 [23, 119]—a mismatch that again allowed an attacker to hijack the browser renderer's control flow.

A constellation of other factors make writing range analysis routines even harder. For example, JITs internally distinguish 32-bit integer values from double-precision floats. This is crucial for performance.[1] But this also means that the range analysis must correctly determine whether an output can be within the range of possible 32-bit numbers—for the Firefox range object, whether the fields `hasInt32LowerBound` and `hasInt32UpperBound` should be set. As another example, since JIT speed directly affects users' experience, the range analysis must be *fast*—it must run just in time—and usefully *precise*—it must produce information useful enough to assist other optimization passes (e.g., DCE, RCE, and bounds-check elimination (BCE)). To this end, JIT developers implement range analysis in low-level languages like C++ and eschew verbose, readable code for fast, terse code. They also explore the trade-offs between speed and precision: Firefox, for example, tracks integer ranges precisely—by tracking lower bounds and upper bounds—but approximates wider floating-point ranges by only tracking exponents (in addition to tracking special values).

2.4 Using VeRA to Express Range Analysis

VeRA is a subset of C++ for writing verified range analysis routines. If browser developers write their analysis logic in VeRA C++, they can compile it into automatic correctness proofs for custom properties. We ported twenty-two Firefox routines to VeRA C++; Figure 2 shows Firefox's implementation of range analysis for the right shift operator, while Figure 3 shows the VeRA version. To calculate the range of possible output values for >>, it masks constant `c` with 31, per JavaScript semantics [57]. Then, it shifts the left-hand

[1]This makes it possible for the JIT to efficiently compile array lookups into a direct memory accesses (as opposed to lookups in a hash map) [111]

operand's lower and upper bound by the masked constant. If `rsh` is given an `lhs` range of [10,100] and a `c` constant of 2, it will return a range of [2, 25].

2.5 Using VeRA to Verify Range Analysis

VeRA also provides an internal DSL for expressing the semantic meaning of each computed fact. Given a range analysis dataflow fact R, its semantic meaning is a predicate $[\![R]\!]$ over values. For expository purposes, the following is a simplified semantic meaning for ranges over 32-bit integers:

$$[\![R]\!](v) \triangleq \ R.\texttt{lower} \leq v \leq R.\texttt{upper}$$

The actual semantic meaning we use in practice (described in Section 5) is far more complicated since it includes floating point numbers, special values, and implementation-specific Firefox invariants.

Once the semantic meaning of range facts is defined, the verification condition can be described as follows. We show the case for a binary operator `op`, but a similar approach can be used for other kinds of operators. We use $\texttt{op}_{ra}$ to denote the range analysis flow function for `op` (where "ra" stands for "range analysis"). This flow function, implemented in a subset of C++, takes two range objects and returns a resulting range object. Finally, we use $\texttt{op}_{js}$ to denote the JavaScript semantics of `op`.

The verification condition for `op` is defined as follows (where ranges R_1, R_2, and JavaScript values v_1, v_2, are universally quantified):

$$[\![R_1]\!](v_1) \wedge [\![R_2]\!](v_1) \Rightarrow [\![\texttt{op}_{ra}(R_1, R_2)]\!](\texttt{op}_{js}(v_1, v_2))$$

This condition states that the flow function for `op` "preserves" the semantic meaning of range facts: if the semantic meaning of the incoming range facts holds on certain incoming values to the operator, then the semantic meaning of the output range fact will hold on the value produced by the JavaScript semantics of the operator on those values.

Let's apply this definition to the `rsh` flow function from Figure 3. In this case `op` is >> ; $\texttt{op}_{ra}$ is `rsh` ; and $\texttt{op}_{js}$ is $>>_{js}$. While >> is a binary operator, the version of the operator described in Figure 3 is the one where the second parameter is a constant, and so we only need to consider range inputs for the first parameter. As result, we have the following verification condition (where range R and JavaScript values v, n are universally quantified):

$$[\![R]\!](v) \Rightarrow [\![\texttt{rsh}(R, n)]\!](v >>_{js} n)$$

Once a verification developer specifies the predicate $[\![R]\!]$, VeRA automatically proves it (using an SMT solver) for each range analysis function. The browser developer need only write the flow functions using a familiar subset of C++; Figure 3 is an example of such a flow function. The above is intended to give a sense for how verification works in VeRA, but the details are described in Section 5.

Feature	Syntax
Declarations	
Classes	`class MyClass {}`
Variables/Fields	`uint32_t x = 4;`
Functions/Methods	`uint32_t f() {}`
Statements	
If/Else	`if (expr) {} else {}`
Assignment	`a = b;`
Void Call	`func(a, b);`
Return	`return foo;`
Expressions	
Casting	`(uint16_t) expr`
Member Access	`s->m;`
Function Calls	`func(a,b)`
Numeric Literals	`1234`, `0xffff`, `47.0`
Comparison	==, !=, >=, >, <=, <
Binary Ops	+, -, *, /, &, |, ^, >>, <<
Unary Ops	~, !, -

Figure 4. C++ constructs that VeRA supports

3 VeRA C++

In this section, we describe the programming language that browser developers use to implement range analysis flow functions in VeRA. This language is called VeRA C++, and is a subset of C++. As shown in Figure 4, various standard C++ features are not included in VeRA C++, the most notable of which may be loops (including recursion). Looping would make the verification much more complicated, because it would require determining loop invariants. In practice, though, omitting looping constructs does not affect VeRA C++'s expressiveness, since flow functions for realistic range analysis don't rely on loops; for example, none of Firefox's analysis functions use loops. Note that even though there is no looping allowed *inside* a flow function, iteration *does* occur in the dataflow analysis algorithm that finds a solution to the flow functions: indeed, that algorithm applies flow functions repeatedly until reaching a fixed point.

To allow programmers to describe the data structures for their range analysis, VeRA supports C++ classes. For example, the VeRA C++ code in Figure 3 uses a `range` class, which is a modified version of the `Range` class from the Firefox codebase (Figure 1). Finally, since VeRA C++ is a subset of C++, all VeRA C++ range analyses can be directly incorporated into the Firefox codebase once they are automatically verified.

4 Translating VeRA C++ to SMT

Before we describe verification conditions (Section 5), we show how to translate VeRA C++ programs to SMT. We describe the challenges with C++ to SMT translation, and then describe how we address theses challenges; Section 7 explains why we choose this particular approach.

Feature	Operations
Cast	`(type)`
Conditional Ternary	`? :`
C++ Comparison	==, !=, >=, >, <=, <
C++ Binary Ops	+, -, *, /, &, |, ^, >>, <<
C++ Unary Ops	~, !, -

Figure 5. VeRA IR operations

4.1 Challenges in Compiling C++ to SMT

One challenge in compiling C++ to SMT is that the theories defined in SMT solvers [37] provide clients with low-level operators like logical right shift, bitwise and, and assignment; they do not provide high-level C++ control-flow constructs like branches and functions. Instead, the compiler must build these constructs out of lower-level SMT primitives.

The next challenge when translating C++ to SMT is that the semantics of SMT types and operators are different from their C++ counterparts. For example, translating the C++ right-shift operator directly to the raw SMT right-shift operator would be incorrect because it does not take into account the subtle interaction of sign bits and the difference between logical and arithmetic shifts. Furthermore, VeRA verification uses the theory of fixed-sized bit-vectors, and bitvectors do not come with a notion of sign; instead, for each operator (e.g., C++'s less-than operator), VeRA must choose a corresponding SMT operator (e.g., signed or unsigned comparison) depending on the C++ typing context. Some C++ operators work on both integer and floating point types (e.g., +), but this is not the case in SMT. Thus, VeRA must also choose which version of the operator (e.g., floating-point or bitvector addition) to select based on the operator's input types.

One final challenge is that the C++ standard includes undefined behavior—instruction and operand combinations whose behavior is explicitly *not* prescribed by the standard [29]. For example, the compiler is allowed to replace undefined "`x / 0`" with anything. In contrast, the functions defined by the theory of bit-vectors are *total functions*, i.e. there is a well-defined result for all inputs, and they are free from side effects. Thus, a translation from C++ to SMT should not accidentally ascribe SMT's well-definedness to C++.

4.2 Overcoming Challenges with an IR

Because of the differences between C++ and SMT, we create an Intermediate Representation (IR) that sits between the two languages. The compilation from C++ to IR takes care of rewriting control flow constructs like branches, function calls, and return values. The translation from IR to SMT takes care of the gap in semantics between the languages' operators and types.

The IR consists of assignment statements in SSA form. The right-hand side of each assignment is an expression tree. Each node in the tree is labeled by an IR operator, and has child nodes that represent the operator's parameters. Figure 5 outlines the operators in the IR.

Compiling C++ to IR. We compile VeRA C++ into control-flow-free IR using a series of transformations. The transformations must eliminate if statements, return statements, function calls, and method calls, and must alter variable assignments to satisfy SSA (which the IR expects).

The transformation to SSA is standard. To handle control flow, VeRA re-writes if statements into straight-line code using predicated execution. For example,

```
if (c) { x = 1; } else { x = 2; }
```

becomes:

```
x_1 =  c ? 1 : x_0; x_2 = !c ? 2 : x_1;
```

VeRA handles function calls using inlining, which works well because VeRA C++ disallows recursion. Thus, each call to a function gets its *own* set of assignments; calling "`foo(1); foo(2);`" will generate two separate, versioned copies of `foo`. The rewrites for method calls, return values, nested conditionals, and returns from within conditionals are all similar to the above translations.

Compiling IR to SMT. Compiling the IR to SMT requires some additional information to be stored in the IR. In particular, each IR node will include the following three pieces of information (in addition to an operator and children):

1. The SMT term generated for this IR node
2. The C++ type of this node (e.g., `int32` or `double`), which is used to generate the correct version of SMT operations
3. An "undef" bit, which indicates whether the node is the result of an operation with undefined behavior. For example, in the statement `x = 4 << -1`, `x`'s undef bit would be set since a left-shift by a negative value is undefined.

This information, computed during the translation from IR to SMT, is critical for generating SMT terms that faithfully replicate the C++ semantics of the original VeRA C++ code.

We compile IR to SMT as follows. First, we translate assignment nodes into equality, which is seamless because the IR is already in SSA form. Second, we translate the expression tree that appears on the right-hand side of each assignment using a bottom-up traversal that simultaneously computes the SMT term, the C++ type, and the "undef" bit (the three pieces of information described above).

As an example, consider a comparison operator. If either argument is unsigned, the compiler generates an unsigned SMT comparison; otherwise, it generates a signed one.

As another example, consider an IR left-shift `a << b`. If `a` is unsigned, the generated SMT term is the logical left-shift of `a` by `b`; the result type is unsigned; and the undef bit is equal to `undef(a)` ∨ `undef(b)` ∨ `isNegative(b)`.

If `a` is signed, the generated SMT term is again the logical left-shift of `a` by `b`, and the result type is unsigned. However, the "undef" bit is more complicated, because C++ 14 dictates that `a << b` has undefined behavior if the shift operation discards any bits of `a` that were set. To detect discarded bits in the 32-bit case, we perform the shift in 64-bits (`(int64)a << (int64)b`), and then check if any of the high 32 bits of the result are set. If so, some bit was shifted off the end of `a`, and the operation has undefined behavior. As a result, the "undef" bit is equal to: $\texttt{undef}(a) \lor \texttt{undef}(b) \lor \texttt{isNegative}(b) \lor (a <<_{64} b)[63:32] \neq 0_{32}$.

5 Verification

In this section, we describe how VeRA proves range analyses correct; dataflow analysis "correctness" means that the facts concluded by the analysis are correct with respect to the semantics of the program. For range analysis: if the analysis concludes that a certain variable is in a range, the analysis is correct if the variable is *actually* within that range according to the program's semantics. Note that we only consider correctness of range analysis facts, not their precision.

People typically prove dataflow analyses correct through abstract interpretation [53]. In abstract interpretation, we start by setting up a connection between computed dataflow facts and the semantics of the program. In our setting, we do so by defining, for each range fact R that is computed about a variable, a *semantic meaning*—a predicate $[\![R]\!]$ over runtime values. This predicate establishes the connection between the computed dataflow facts and the semantics of the program: we say that a range fact R on a variable x is *correct* at a program point if, for all possible values v that x can take at runtime, we have $[\![R]\!](v)$.

The goal of our verification is to establish that the implemented range analysis is *correct*, meaning that when the dataflow analysis algorithm terminates, all facts it has computed are correct. In abstract interpretation, this is achieved by proving *local preservation* conditions on all flow functions. These conditions can be shown to imply that the entire range dataflow analysis is correct [53]. It is these local preservation conditions that we ask an SMT solver to prove.

Recall that we use $\texttt{op}_{ra}$ to denote the range analysis flow function for an operator `op` (which for simplicity we assume to be a binary operator). Recall also that we use $\texttt{op}_{js}$ to denote the JavaScript semantics of `op`. As mentioned in Section 2, we define the *local preservation* condition for an operator `op` as:

$$[\![R_1]\!](v_1) \land [\![R_2]\!](v_1) \Rightarrow [\![\texttt{op}_{ra}(R_1, R_2)]\!](\texttt{op}_{js}(v_1, v_2))$$

This condition states that the flow function for `op` "preserves" the semantic meaning of range facts: if the incoming range facts are correct, then the propagated range fact is correct.

In this section, we first define the semantic meaning $[\![R]\!]$ of a given range fact R. Then, we discuss how VeRA translates $\texttt{op}_{js}$ to SMT (since Section 4 explains translation of $\texttt{op}_{ra}$), and talk about how proofs work in practice.

$\texttt{inRange}(R, v) \triangleq$

$$
\begin{array}{lr}
R.\texttt{exp} < \texttt{e\_INF} \implies \neg\texttt{isInf}(v) & \text{(R1)} \\
\land R.\texttt{exp} \neq \texttt{e\_INF\_OR\_NAN} \implies \neg\texttt{isNaN}(v) & \text{(R2)} \\
\land \neg R.\texttt{canBeNegZero} \implies v \neq -0.0 & \text{(R3)} \\
\land \neg R.\texttt{canHaveFraction} \implies \texttt{round}(v) = v & \text{(R4)} \\
\land R.\texttt{hasInt32LowerBound} \implies (\texttt{isNaN}(v) \lor v \geq R.\texttt{lower}) & \text{(R5)} \\
\land R.\texttt{hasInt32UpperBound} \implies (\texttt{isNaN}(v) \lor v \leq R.\texttt{upper}) & \text{(R6)} \\
\land R.\texttt{exp} \geq \texttt{expOf}(v) & \text{(R7)}
\end{array}
$$

Figure 6. The definition of the predicate $\texttt{inRange}(R, v)$ that states whether a floating-point number v is in range R.

$\texttt{wellFormed}(R) \triangleq$

$$
\begin{array}{lr}
R.\texttt{lower} \geq \texttt{JS\_INT\_MIN} \land R.\texttt{lower} \leq \texttt{JS\_INT\_MAX} & \text{(W1)} \\
\land R.\texttt{upper} \geq \texttt{JS\_INT\_MIN} \land R.\texttt{upper} \leq \texttt{JS\_INT\_MAX} & \text{(W1)} \\
\land \neg R.\texttt{hasInt32LowerBound} \implies R.\texttt{lower} = \texttt{JS\_INT\_MIN} & \text{(W2)} \\
\land \neg R.\texttt{hasInt32UpperBound} \implies R.\texttt{upper} = \texttt{JS\_INT\_MAX} & \text{(W2)} \\
\land R.\texttt{canBeNegZero} \implies \texttt{contains}(0, R) & \\
\land (R.\texttt{exp} = \texttt{e\_INF} \lor R.\texttt{exp} = \texttt{e\_INF\_OR\_NAN} \lor R.\texttt{exp} \leq 1023) & \text{(W3)} \\
\land (R.\texttt{hasInt32LowerBound} \land R.\texttt{hasInt32UpperBound}) & \\
\quad \implies R.\texttt{exp} = \texttt{expOf}(\texttt{max}(|R.\texttt{lower}|, |R.\texttt{upper}|)) & \\
\land R.\texttt{hasInt32LowerBound} \implies R.\texttt{exp} \geq \texttt{expOf}(R.\texttt{lower}) & \text{(W4)} \\
\land R.\texttt{hasInt32UpperBound} \implies R.\texttt{exp} \geq \texttt{expOf}(R.\texttt{upper}) & \text{(W4)}
\end{array}
$$

Figure 7. Well-formedness for a Firefox range.

5.1 Semantic Meaning of Predicate Facts

We split the semantic meaning $[\![R]\!]$ of a range fact R into two parts: (1) a predicate that connects R to values and (2) a well-formedness invariant expressed solely on R. In particular:

$$[\![R]\!](v) \triangleq \texttt{inRange}(R, v) \land \texttt{wellFormed}(R)$$

The `inRange` predicate connects the range fact to values from JavaScript semantics. The `wellFormed` predicate is more unique: it isn't about a connection to semantic values, but instead about implementation-specific Firefox invariants that are necessary for proving the flow functions correct. We explain each piece of $[\![R]\!]$ in turn below.

The* `inRange` *predicate. Figure 6 shows the `inRange` predicate, which we derived from Firefox code and comments. An important point here is that this predicate is unusually complex—far more complex than any of the semantic meanings used in prior automated verification efforts for program analyses [82]. A typical meaning for a range analysis fact in prior work is:

$$\texttt{inRange}(R, v) \triangleq R.\texttt{lower} \leq v \leq R.\texttt{upper}$$

This predicate is simpler than our `inRange` because `inRange` must handle the complexities of realistic range analysis for JavaScript (§2), i.e., tracking floating point numbers, since all JavaScript values are double-precision floats.

Given a range fact R and a JavaScript value v, the predicate `inRange`(R, v) is true iff the following conditions hold. First, if the exponent field of R is less than a special Firefox value, `e_INF`, v must not be infinite. Similarly, if R's exponent is less than the special value `e_INF_OR_NAN`, v must not be `NaN`. If R's `canBeNegativeZero` flag is not set, v should not be -0.0; if R's `canHaveFractionalPart` flag is not set, v should be a whole number. Finally, if R has a lower bound (`hasInt32LowerBound` flag), v should be greater than or equal to that lower bound (`lower`). The `hasInt32LowerBound` flag indicates whether a range contains numbers that are smaller than thirty-two bit integers; it allows Firefox to internally use integers to represent JavaScript numbers when possible.

The* `wellFormed` *predicate. Figure 7 shows the `wellFormed` predicate, which is unusual because it does not relate the range fact to semantic values. Instead, it simply imposes constraints on the range fact itself; it is an invariant that Firefox flow functions depend on to be correct. We derived this invariant from a set of long comments and a handful of invariant-checking functions in the Firefox codebase.

All range facts should have `canBeNegativeZero` set only when zero is contained within their range[2], where `contains` is defined as follows for range R and value v:

$$\texttt{contains}(R, v) \triangleq v \geq R.\texttt{lower} \land v \leq R.\texttt{upper}$$

Furthermore, exponents should either be in the range 0 to 1023, or should have the special value `e_INF` (1024) for infinity, or the special value `e_INF_OR_NAN` (65535) for `NaN` or infinity. In addition, the exponent should also be consistent with the lower and upper bound, if they exist.

Now, we walk through an example showing how Firefox routines break when invariants are violated, focusing on the invariant relating `hasInt32LowerBound` with R`.lower` and `hasInt32UpperBound` with R`.upper`. The Firefox implementation of range analysis requires that if a range's `hasInt32LowerBound` is unset, then its `lower` field equals `JS_INT_MIN`; similarly, if its `hasInt32UpperBound` is unset, its `upper` field equals `JS_INT_MAX`. Consider the following Firefox range analysis code for `max`, which calculates the new `lower` field and the new `hasInt32LowerBound` flag in the returned range fact:

```
int32_t newLower = max(lhs->lower, rhs->lower)
bool newHasLower = lhs->hasInt32LowerBound ||
↪  rhs->hasInt32LowerBound
```

Now consider two input ranges to the `max` function, R_1 and R_2, and assume the output range is R_3. If the Firefox invariant doesn't hold, we can have $\neg r_1$`.hasInt32LowerBound` and R_1`.lower` $= 1000$, and R_2`.hasInt32LowerBound` and R_2`.lower` $= -1000$. This means that the above code will set R_3`.hasInt32LowerBound` to true, and R_3`.lower` to 1000 (since the result lower bound is the maximum of the input lower bounds).

Now consider two JavaScript values v_1 in R_1 and v_2 in R_2 (meaning that `inRange`(R_1, v_1) and `inRange`(R_2, v_2) are both true). Let v_1 be $-\infty$ (since $\neg R_1$`.hasInt32LowerBound`) and v_2 be -1000. Then, JavaScript semantics says that $\texttt{max}_{js}(v_1, v_2)$ is -1000, but -1000 is well below R_3's lower bound of 1000.

However, once we add the `wellFormed` invariant, this problem is fixed: when `hasInt32LowerBound` is not set, `lower` must be `JS_INT_MIN`, so R_1`.lower` cannot be 1000. Now that R_3's lower bound is `JS_INT_MIN` instead of 1000, it correctly captures $\texttt{max}_{js}(v_1, v_2)$ by including -1000. This example shows that `max`'s range analysis *relies* on the invariant that `lower` is `JS_INT_MIN` when `hasInt32LowerBound` is not set.

Other Verification Conditions. We prove two additional properties of the Firefox range analysis: we prove the correctness of the functions for combining range facts—union and intersection—which the JIT uses within its larger range analysis loop. We will use `union` and `intersection` to refer to Firefox's union and intersection functions for range facts. Conceptually, given two range facts R_1 and R_2, we want to show that `union`(R_1, R_2) is an overapproximation[3] of the mathematical union operation $\cup$ on the semantic values contained in R_1 and the semantic values contained in R_2. We achieve this as follows. Given two well-formed ranges R_1 and R_2, we let $R_3 = \texttt{union}(R_1, R_2)$. Then we want to show that for all JavaScript values v, we have:

$$\texttt{inRange}(R_1, v) \lor \texttt{inRange}(R_2, v) \implies \texttt{inRange}(R_3, v)$$

Similarly, for intersection we let $R_3 = \texttt{intersect}(R_1, R_2)$, and prove that:

$$\texttt{inRange}(R_1, v) \land \texttt{inRange}(R_2, v) \implies \texttt{inRange}(R_3, v).$$

5.2 Using VeRA to Express Predicates

VeRA exposes an internal verification domain-specific language (DSL) embedded in Haskell. Verification developers use the language to express verification infrastructure, including verification conditions and the semantic meaning of range facts. The DSL exposes a number of JavaScript operators—i.e., implementations of $\texttt{op}_{js}$—against which to verify range analysis functions. If verification developers wish to verify new operations, it is straightforward to expose new JavaScript routines in the VeRA internal DSL. To express predicates and verification conditions, the DSL also exposes SMT directives. They allow verification developers to make assumptions, call the SMT solver, and push and pop new incremental solver contexts.

In practice, proving the conjunction of `wellFormed` and `inRange` for each operator is suboptimal: if any component of either predicate does not finish, the entire verification

[2]This is only an invariant on optimized ranges; unoptimized may have set flags even if their ranges don't contain zero.

[3]And it is necessarily an overapproximation given how Firefox's range analysis object is implemented: consider a union of two ranges, one that can include fractions and one that can't.

proof does not finish. As a result, we prove individual conditions within each predicate separately; Section 6 describes each condition that we prove and the time the proof takes to finish (or not). Mechanically, to prove a given predicate part, we query an SMT solver with its negation—e.g., we check whether the operation can produce any values *outside* of the computed range. If the formula is unsatisfiable, there are no such values and the range analysis routine is safe. If the formula is satisfiable, the model provided by the solver is a concrete counterexample that captures that input ranges and values for which the range analysis routine is unsafe.

6 Implementation and Evaluation

We implement VeRA—both the compiler for VeRA C++ and the internal verification language—in 1384 lines of Haskell. Since the implementation details of the compiler are standard, we only describe the details relevant to answering our evaluation questions. All our source code and data sets are available at https://vera.programming.systems.

We evaluate VeRA by asking six questions:

Q1 Can VeRA prove Firefox range analysis routines correct?
Q2 Can VeRA proofs catch real correctness bugs?
Q3 Are the VeRA proofs correct?
Q4 Do the verified routines work correctly in Firefox?
Q5 How do the verified routines perform in Firefox?
Q6 How hard is it to integrate verified routines into Firefox?

To answer these questions, we port 21 top-level Firefox range analysis flow functions to VeRA C++, try to prove their correctness, and then re-integrate them into the browser. We are able to: prove 137 separate facts about these routines; identify a new Firefox analysis bug; and correctly detect an old analysis bug. A version of Firefox that uses our verified routines still performs comparably to standard Firefox, and it still passes *all* (147,322) Firefox JavaScript tests.

6.1 Proofs

We choose to verify 19 Firefox flow functions because they are the complete set of Firefox Range-type flow functions for JavaScript operators; we discuss this further in Section 7 (e.g., as a result, we don't check division). In addition, we verify the union and intersect functions, which are not JavaScript operators but instead Firefox-internal functions that combine two different Range objects; this brings the total number of routines we attempt to verify to 21. Verification of both union and intersect times out, but VeRA finds a bug in our port of an older, broken version of intersect [9]. Finally, we port 25 helper functions called by each verified function—i.e., every function except the optimize function (§7). Figure 8 summarizes our results.

All operators followed by asterisks in the table (e.g., rsh) are only valid for 32-bit ranges.[4] Thus, for each bitwise operator, we only prove (1) the simple predicate from Section 2 and (2) the absence of undefined behavior in the result range's upper and lower bounds. We don't worry about wellFormed for these operations, either; none of them alter the floating-point-specific fields of the range.

For floating-point operators (e.g., add), we separately prove each condition in the inRange predicate and the wellFormed predicate with two exceptions: since we do not call optimize on our result ranges, we only verify the wellFormed conditions that apply to non-optimized ranges (i.e., our output ranges are correct but may not have the tightest possible bounds). Figure 8 goes into more detail about which columns correspond to which conditions in Figure 6 and Figure 7.

Multiple proofs fail: ceil and broken intersect are both real bugs, while ursh and ursh' require extra invariants on the input ranges. We don't amend them because their failure is actually an interesting manifestation of a comment above both functions [26]:

```
// ursh's left operand is uint32, not int32, but for range
// analysis we currently approximate it as int32. We assume
// here that the range has already been adjusted...
```

In other words, over all possible inputs, ursh is *not* correct.

Our proof code, i.e., the code that automatically verifies each part of both the inRange and wellFormed predicates, amounts to 804 lines of Haskell. We run all proofs on a virtual machine (QEMU-KVM, Linux 5.3.11) running Arch Linux with 16 GB of memory and 4 vCPUs. The processor is an AMD Ryzen Threadripper 2950X 16-Core with a base clock rate of 3.5GHz. We use the Z3 SMT solver (4.8.6), which we call with custom Haskell bindings that extend the haskell-z3 library [30], using a 20-minute timeout.

Can we prove Firefox range analysis routines correct? We successfully prove or refute 137 conditions out of a possible 159, for a success rate of ≈86%; the shortest proofs complete in under a second, while the longest takes ≈ten minutes. The results suggest that **R5.double**, **R6.double**, and **W4** are particularly challenging to verify. **R5.double** and **R6.double** are more challenging than their integer counterparts because they involve reasoning about floating-point values, which is generally more expensive. **W4** is challenging because it involves proving a relationship between two properties of the range, both of which may be modified by the range analysis. Finally, **R1** and **W3** of Math.ceil may timeout because they involve bounding the size of an exponent, since Math.ceil involves extracting the exponent from the absolute value of the range bounds.

There is hope, however. Though VeRA does not verify conditions for certain routines (e.g., correctness of intersect),

[4]This is an internal Firefox invariant: each one of these functions starts with an assertion that its operands ranges only include 32-bit numbers.

Operation	**R1**	**R2**	**R3**	**R4**	**R5.i32**	**R5.double**	**R6.i32**	**R6.double**	**R7**	**W1**	**W2**	**W3**	**W4**	**Undef**
`add`	15	2	5	386	2	∞	2	∞	21	2	1	2	80	1
`sub`	13	2	11	445	8	∞	5	∞	14	2	2	2	78	1
`and∗`	-	-	-	-	2	-	1	-	-	-	-	-	-	1
`or∗`	-	-	-	-	2	-	2	-	-	-	-	-	-	1
`xor∗`	-	-	-	-	2	-	2	-	-	-	-	-	-	1
`not∗`	-	-	-	-	2	-	1	-	-	-	-	-	-	1
`mul`	92	65	22	362	∞	∞	∞	∞	94	4	4	4	∞	11
`lsh∗`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`rsh∗`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`ursh∗`	-	-	-	-	**X**	-	**X**	-	-	-	-	-	-	1
`lsh'∗`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`rsh'∗`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`ursh'∗`	-	-	-	-	**X**	-	**X**	-	-	-	-	-	-	1
`abs`	1	1	1	5	1	∞	1	224	1	1	1	4	∞	1
`min`	2	20	2	2	5	224	2	∞	3	2	1	2	∞	1
`max`	3	17	2	2	15	∞	2	∞	4	3	2	12	∞	1
`floor`	4	2	1	5	1	146	1	9	54	1	1	5	∞	1
`ceil`	∞	1	**X**	8	1	5	1	9	266	1	1	∞	∞	1
`sign`	1	1	1	1	1	2	1	2	1	1	1	1	2	1

Figure 8. The time it takes, in seconds, for VeRA to verify predicates the predicate from Section 5. **R1**-**7** correspond to the lines in the `inRange` predicate, while **W1**-**4** correspond to line groups one through four in `wellFormed`. Broadly, **R1** makes sure the routine handles infinities correctly, **R2** NaNs, **R3** -0.0, **R4** fractions, **R5**s lower bounds (over both 32-bit integer and double values), and **R6**s upper bounds. **W1** ensures missing upper and lower bounds imply a minimum or maximum value for `lower` and `upper`; **W2** ensures `lower` and `upper` are always valid JavaScript 32-bit numbers; **W3** ensures the range's exponent is valid, and **W4** ensures the exponent is consistent with the upper and lower bound. Finally, **Undef** shows the time it takes to verify that the range computations for `upper` and `lower` are free of undefined behavior. ∞indicates timeout, while **X** indicates verification failure.

it *is* able to catch multiple bugs in broken versions of unverified routines. For example, as we discuss later in this section, we port an older, broken version of `intersect` to VeRA C++. VeRA is able to detect the bug in this version in 173 seconds, even though the proof for the current version of `intersect` never finishes. VeRA also catches a number of errors that we introduced while copying code over from Firefox; as we improved VeRA, it went from being a custom DSL to a subset of C++, so porting was not always as easy as copy-pasting browser code. For example, we switched an upper and lower bound in `sub`, which caused it to fail the floating-point lower bounds check—even though this check never verifies in the fixed version of `sub`. VeRA caught one other porting error in `sub` (use of the field `canBeNegativeZero` instead of the function `canBeZero`), and at least two more in `mul` (switched `lhs` and `rhs`, and use of `canBeFiniteNonNegative` in place of `canBeFiniteNegative`).

A new Firefox bug. VeRA found a bug in Firefox's range analysis for the `Math.ceil` operator [4], which rounds its input up to the nearest integer (e.g., the ceiling of 2.5 is three). The bug, which follows, has existed since the routine's introduction six years ago [2]:

```
Range* Range::ceil(TempAllocator& alloc, const Range* op) {
  Range* copy = new (alloc) Range(*op);
  if (copy->hasInt32Bounds())
    copy->max_exponent_ =
  ↪   copy->exponentImpliedByInt32Bounds();
  else if (copy->max_exponent_ < MaxFiniteExponent)
    copy->max_exponent_++;

  copy->canHaveFractionalPart_ = ExcludesFractionalParts;
  copy->assertInvariants();
  return copy;
}
```

The routine looks straightforward. Given an input range, it adjusts the range's exponent upwards by one—to account for upward rounding—and unsets the `canHaveFractionalPart` flag—since the result is always a whole number.

The problem lies in what `ceil` *doesn't* do: it never adjusts the input range's `canBeNegativeZero` flag. JavaScript semantics, though, dictate that `Math.ceil(x) = -0` when x is between zero and negative one. Therefore, given a range with lower and upper bound [-1, 0] and an unset `canBeNegativeZero` flag, `ceil` will *not* correctly set the flag.

VeRA identifies the error after about three seconds, and provides the following (shortened) counterexample:

```
result_range_canBeNegativeZero : 0
start_range_canBeNegativeZero : 0
start_range_lower : -128
start_range_upper : 0
start : -1.166614929399505e-301
```

It finds a start value, `start`, within the `start_range` of [-128, 0] with the `canBeNegativeZero` flag unset. It notes that `ceil` of `start` is `result`, -0, but that the result range does not have the `canBeNegativeZero` flag set.

After confirming the bug with Mozilla engineers, we tried to patch the bug and use VeRA to verify the patch. VeRA rejected our first attempt—it was wrong—but approved the next one [4]:

```
copy->canBeNegativeZero_ = ((copy->lower_ > 0) ||
↪  (copy->upper_ <= -1))
                                ? copy->canBeNegativeZero_
                                : IncludesNegativeZero;
```

Now, the function sets the `canBeNegativeZero` flag to be true when the resulting range can be includes values between negative one and zero.

An old Firefox bug. To test VeRA further, we port a buggy version of Firefox's `intersect` operator, which takes the intersection of two ranges [10]. The buggy operator deduces that the intersection of two ranges $r_1 \cap r_2 = \emptyset$ if the upper bound of r_1 doesn't overlap with the lower bound of r_2. This behavior is correct—unless both ranges contain NaN, in which case the result range should include NaN, too. VeRA identifies the error after 173 seconds, and provides a counter example showing an element in the two input ranges (NaN) that was not included in the output range.

Are VeRA proofs correct? Verifiers can be as broken as the code they are intended to verify. To guard against this possibility, we use Haskell's QuickCheck [52] to automatically random-test the semantics of (1) the operators in the VeRA IR and (2) the JavaScript operators that VeRA uses for verification.[5] For each JavaScript operator, we generate JavaScript code that performs the operations, evaluates it with Node.js, and compares the result against that produced by our SMT model. We use Node.js (version 10.1.0) because it uses the Chrome V8 JavaScript engine and is thus likely to have different bugs from Firefox (and thus VeRA). Our C++ operator tests are similar; we use Clang version 9.0.0. This checking proved useful—e.g., QuickCheck found a bug in our implementation of the C++ floating-point `abs` operator. For further assurance, we also cross-checked our JavaScript semantics against KJS [96].

6.2 Verified Routines in the Browser

We integrate our verified range analysis into Firefox 72.0a1 (commit 10c8c9240d). Our versions of the Firefox range analysis functions amount to 621 lines of C++ code. In this section, we describe the effort it took to retrofit Firefox and measure the performance of our modified browser when compared with *vanilla*, unmodified Firefox.

[5]We use QuickCheck 2.13.2 on GHC 8.6.5 and configure it to test each operator on different types (e.g., integers of varying widths) 1,000 times.

How hard is it to integrate VeRA code into Firefox? We retrofit Firefox in two steps. First, we extend Firefox's Range class with a `verifedRange` field—pointing our verified object—and modify the class setters and getters to forward all accesses to the corresponding `verifiedRange` fields. For example, we rewrite,

```
Range* Range::abs(TempAllocator& alloc, const Range* op) {
  int32_t l = op->lower_;
  int32_t u = op->upper_;
  ...
}
```

to:

```
Range* Range::abs(TempAllocator& alloc, const Range* op) {
  int32_t l = op->verifiedRange.lower;
  int32_t u = op->verifiedRange.upper;
  ...
}
```

Then, we replace individual function bodies with calls to our verified functions. For example, we rewrite `abs` to:

```
Range* Range::abs(TempAllocator& alloc, const Range* op) {
  auto vRange = verified::abs(op->verifiedRange);
  return Range::fromVerifiedRange(alloc, vRange);
}
```

Our porting effort was surprisingly low: two engineers—neither of whom is a Firefox core developer—integrated our VeRA C++ routines into Firefox over the course of two days. We think this effort can be reduced even further: both steps are mechanical and can be automated to only require human intervention when tests fail.

Do the verified routines work correctly? To test our ports, we run all of Firefox's JavaScript suites (using their `mach` build tool): the 7,364 JIT tests, 394 JSAPI tests (which use the `Range` interface directly), and 139,564 general JavaScript tests. Though VeRA passes all of Firefox's tests now, it failed some tests along the way.

After altering Firefox to call VeRA routines whenever applicable, all but three of the JIT tests passed; one timed out and two failed. This was because one of our ported functions did not initialize every field of the range object; the function was only called by a range analysis routine for a bitwise operator, so to verify it, we did not need to set the floating-point-specific range fields. We also failed two of the JSAPI tests, both due to typos in our porting of `intersect` and `sign` functions. The VeRA verification actually caught the `sign` bug, but due to a miscommunication within the team, the fix didn't make it into Firefox immediately. The bug in `intersect` caused it to over-approximate the possibility of negative zeroes—but our verification specifically allows over-approximation by design (§5).[6] Once we fixed these bugs, all JIT and JSAPI tests passed, and when we ran the other 139,564 Firefox JavaScript tests, all of those passed, too.

[6]Range combinations in Firefox are necessarily over-approximations.

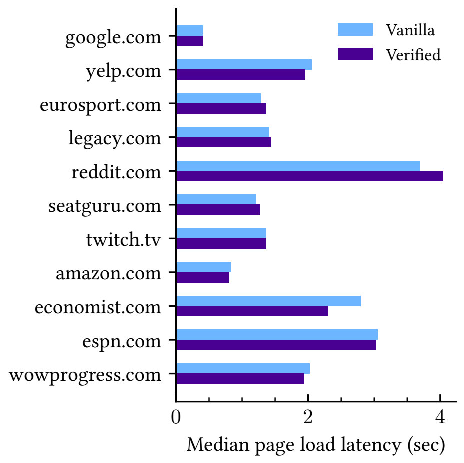

Figure 9. Page load latency of popular and unpopular websites.

How do the verified routines perform? We evaluate the performance impact of VeRA on JavaScript execution and end-to-end page latency. We run all the performance benchmarks on an Intel 8 core (i7-6700K machine with a base clock rate of 4.0GHz) machine running Arch Linux with 64 GB of memory. We compare our browser against a vanilla, unmodified Firefox, and find the impact of VeRA to be on par. This is not surprising: our C++ code is very similar to Firefox's original range analysis code.

To measure the performance of VeRA on JavaScript execution, we run the JetStream 2 benchmarking suite [32], created by the WebKit team. This benchmark subsumes the now-deprecated SunSpider [33] benchmarking suite.[7] JetStream 2 consists of 64 benchmarking tests that measure representative JavaScript and WebAssembly workloads for both start-up times, code execution, and "smoothness".[8] Each test reports a *score*, corresponding to how well the browser performed. We present the JetStream 2 results, as run on 2019-11-22, in Appendix A of [48]; the overall performance of our modified Firefox is on-par with unmodified Firefox—our browser scored 75.688 while vanilla Firefox scored 77.206. JetStream 2 computes this score by "taking the geometric mean over each individual benchmark's score". Our mean and median scores are within 3.2% and 8.5%, respectively. The maximum difference in scores is in 52%, in the string-unpack-code-SP benchmark which stresses string manipulation. We think the big differences are largely due to noise; running the string-unpack-code-SP benchmark outside the browser (1,000 iterations in the js shell) we only observed a 0.48% difference.

We measure the impact of our verified code on end-to-end page latency by browsing a representative sample of both popular and unpopular websites. In particular, we use the list of 11 sites curated by the Chrome team in their recent Chrome sandboxing work [102]. For each site, we use Mozilla's Talos benchmarking tool to measure the time it takes to render a page (i.e., the time to first paint [28]), taking the median of 50 runs (after a 5 run warm-up). Figure 9 presents our measurements. The median and average latencies of browsing these sites with our browser are within 5% of vanilla Firefox. The biggest slowdown is on reddit.com (18%), while the biggest speedup is on economist.com (-9%); like [102], we attribute these bigger difference to the inherently noise introduced by media content and the network. Overall, these results are encouraging: VeRA does not impose overheads that are prohibitive to its adoption.

7 Discussion, Limitations and Future Work

Why not existing proof tools? We started out building a DSL for range analysis verification, and ended up building a compiler from both C++ and an internal verification language to SMT. There are many existing tools that can translate programming languages into SMT [49, 100], and they primarily operate on an existing compiler IR (e.g., LLVM IR) instead of defining their own IR. Verifying JavaScript JIT optimization passes at the LLVM IR level is something to strive for in the future, but using a small language is an easier start; to our knowledge, no LLVM-IR-level tool supports JavaScript semantics, some don't support C++ reliably [24], and the ones that do can get lost in complex class object hierarchies—in IR, series of pointer offset calculations—and as a consequence struggle to verify anything at all. Furthermore, using these tools requires integrating them with (very complicated) browser build systems. We, on the other hand, don't provide any guarantees about the final machine code.

What we don't verify. We only verify range analysis routines for Firefox. Chrome's range analysis pass is tied to its type inference pass—different ranges correspond to different types (e.g., 32-bit integers have a bound). We consider extending VeRA to other browsers future work.

Within Firefox, the range analysis functions that we verify all return the basic Firefox `Range` object type. These functions contain local range analysis logic, and are called during the range analysis computation for different MIR nodes, Firefox's middle-level intermediate representation of JavaScript programs. For example, the `computeRange` method for the `MCeil` node (representing `Math.ceil`) simply wraps `Range::ceil` and thus reaps the benefits of our verification effort. Many MIR nodes, however, do not correspond to a JavaScript-level constructs (e..g, `MSpectreMaskIndex` is used to represent a masked array index) and we thus do not verify them. We also do not verify the range analysis algorithm itself nor Firefox's use of ranges in code generation or other optimization passes (e.g., DCE or BCE). These are natural

[7]We did, however, run the SunSpider benchmarks and report the results in Appendix B of [48]. It reported no meaningful performance difference.
[8]The WebAssembly pipeline in Firefox actually uses the JavaScript pipeline and thus VeRA in the verified browser.

extensions to our work. Similarly, since other JIT components (e.g., type inference) have been a source of security vulnerabilities, we hope to address them as future work (e.g., by building on JIT type inference foundations [71, 73]).

8 Related Work

VeRA lies at the intersection of work on compiler verification, JIT compilation, and browser security.

Verifying optimizations using SMT. Various prior systems have used DSLs combined with SMT solvers to express and verify compiler optimization correctness: for example, Cobalt [81], Rhodium [82], PEC [80] and Alive [86]. These systems mostly focus on proving correctness of *transformations*, e.g., scalar optimizations in Cobalt and Rhodium, control flow rewrites in PEC, and peephole optimizations in Alive. There has been much less work on using DSLs and SMT solvers to prove the correctness of *analyses*, with the notable exception of the Rhodium system [82]. While conceptually the techniques in VeRA are similar to Rhodium's techniques for analysis correctness, the analyses in Rhodium are relatively simple, with the semantic predicate of facts often containing a single term. In contrast, VeRA demonstrates how to use DSLs and SMT solvers to verify the correctness of analyses in a realistic setting: our work handles all the corner cases of Firefox's range analysis, which in turn includes a semantic predicate for the range facts with 16 cases.

Another difference between our work and prior work is the type of constructs we support (in VeRA C++). Alive supports pointers and arrays (with static, known sizes), while VeRA does not support either construct—but neither system handles loops, since neither peephole optimizations nor range analysis computations typically require them.

Compiler and analysis verification in a proof assistant. Another approach to general compiler verification is *foundational verification*. In foundational verification, the programmer writes the compiler in a *Proof Assistant*, and then uses the proof assistant to interactively prove that the compiler is correct. Examples of foundationally verified compilers include CompCert [83], CompCertTSO [116], Compositional CompCert [112], and CakeML [79]. Examples of semantic IR frameworks include the Vellvm system [122], which provides a formal semantics for LLVM IR that others can use to verify IR optimizations/transformations.

In addition to entire compilers or IR frameworks, there has also been work on specific analyses and optimizations that are foundationally verified. For example, Versaco [76] is a foundationally verified static analyzer for CompCert; developers can use it to write their own analyses that prove properties of analyzed programs. For specific optimizations, Zhao et al. use Vellvm to verify a version of LLVM's mem2reg transformation, which changes memory references to register references [25, 123]. Mullen et al. use the Coq proof assistant to verify peephole optimizations for the CompCert verified C compiler [83, 90]. Finally, Tatlock et al. extend CompCert with a DSL for expressing optimizations [114]—combining the DSL approach with foundational verification.

Because proofs in foundational verification are performed in full detail, foundational verification provides the strongest possible correctness guarantees. However, these proofs often require a significant amount of expert human guidance, making them very difficult to complete. In contrast, VeRA allows browser developers to express their analysis in a subset of C++, and then provides automated verification to the developer without any additional effort or verification knowledge.

Verification of JIT compilation. There has also been work specifically on verifying correctness of JIT compilers, including work on verifying a non-optimizing JIT compiler [91], and work on defining correctness criteria for trace-based JIT compilation [72]. There is also ongoing work on verifying a JIT in Coq [38], which includes support for some optimizations, though it is not clear which ones; similarly, there is ongoing work on verifying a JIT using symbolic execution [107]. None of this work focuses on the specific challenge that we are addressing, namely verifying the complex analyses that drive optimizations in browser JITs.

Translation validation. Another approach to compiler correctness is translation validation [92, 98, 106]: each time the compiler runs, a validator tries to prove that the transformed code behaves the same as the original code. While translation validation can find compiler bugs, it does not guarantee the absence of bugs, as VeRA attempts to. For translation validation to guarantee the absence of bugs, it would have to do validation on production runs, which incurs compilation overhead—not ideal for a JIT.

Recent work by Taneja et. al. goes further by proposing an algorithm for sound and *maximally precise* dataflow facts like integer ranges (similar to this work) and known bits, among others [113]. For a given code fragment, they (1) use their algorithm (implemented with an SMT solver) to compute dataflow facts about that fragment and (2) compare those facts to the ones LLVM has computed. This technique has identified several precision errors in LLVM's analyses, and adapting it to help developers design tighter ranges is interesting future work.

Verification and floating point numbers. Other systems also handle the challenge of verifying floating-point code. Recently, Boldo et al. formalize IEEE-754 semantics in Coq in order to extend CompCert to support programs that use floating-point numbers [47]. More similar to VeRA, Icing [39], which builds on the verified ML compiler CakeML [79], is a language for writing "fast-math" optimizations. Multiple projects also extend the Alive system to support peephole optimizations related to floating point numbers [89, 94]. Recently, Becker et al. use the Daisy tool [54] to verify optimizations to floating-point computations—in programs, not in compilers—that the Herbie [95] tool generates [40].

Testing compilers. Beyond verification, there are other approaches to compiler correctness. One is automatic testing, or "fuzzing," which finds bugs but does not guarantee the absence of errors. For example, the CSmith tool automatically generates *useful*—well-formed, undefined-behavior-free—inputs for testing C compilers [120]. Dewey et al. present a system for fuzzing the Rust compiler's type checker [56]. Finally, there are fuzzers specific to JavaScript interpreters: Fuzzilli [65, 66], and CodeAlchemist [74]. Fuzzilli has been remarkably effective at finding bugs in JavaScript engines; its bug showcase lists two dozen security issues [11].

Verified JavaScript semantics. There has been a significant effort to formalize (parts of) the JavaScript language. Most of these efforts start with a simple core language and extend it with unwieldy JavaScript features (e.g., `eval`, property descriptors, and **`with`**) [61, 70, 87, 99]. The JSCert JavaScript subset was even mechanized in Coq, from where they extracted a verified correct interpreter [45, 46, 62]. KJS [96] provides a complete JavaScript semantics in the K framework. Though we cross-check our semantics against KJS and JSCert's (where possible), this work is complimentary: they focus on verifying JavaScript semantics, we focus on the JITs that should preserve these semantics.

Verification in the browser. A verified browser kernel, Quark [75], demonstrates that verification is possible in the browser setting. Though we're a long way from a verified Firefox, browsers are interested in verified software. For example, both Firefox and Chrome incorporated verified cryptographic primitives into their TLS stacks [27, 125].

Safer browser JITs. Another approach to JIT safety is to limit the damage of bugs, not prevent them. NaClJIT sandboxes both the JIT compiler and the code it produces [31]. RockJIT applies a control-flow integrity policy to the JIT compiler and the code it produces [93]. NoJITSu prevents code-reuse and data-only attacks [97]. Browser vendors have also modified their JITs to reduce the effectiveness of JIT spraying [85], a technique that allows attackers to introduce bytes of their choice into pages marked executable in browser memory [44].

Large, real-world verification. Finally, beyond what we have discussed so far, there are many other large, real-world verification efforts. The Astrée static analyzer has verified safety properties of Airbus software [55]; the miTLS project provides verified reference implementations of TLS 1.0, 1.1, and 1.2 [42]; Barbosa et. al. [35] give an overview of verification efforts for crypto code, including discussion of an ongoing effort to formally verify the TLS 1.3 protocol. seL4 [78] is a verified operating systems kernel; and various recent efforts have proved properties of systems software like file systems [50, 51, 108] and in-kernel interpreters [118].

9 Conclusion

This paper presents VeRA, a system for verifying the range analysis pass in browser JITs. VeRA allows browser developers to write range analysis routines directly in the browser (in a subset of C++) and provides a DSL that verification developers can use to encode verification properties (e.g., range analysis invariants). VeRA automatically verifies these properties using SMT.

We use VeRA to encode a semantics for Firefox's range analysis, and then port 22 Firefox analysis routines to VeRA C++ in order to verify them. The ported version of the browser performs on-par with the original. Moreover, VeRA detects a bug that has existed in the browser for six years—and verifies the Firefox patch we wrote to fix the bug.

Acknowledgments

We thank the anonymous reviewers for their comments, and especially our shepherd John Regehr for helping us think through both the content and presentation of this paper. Thanks to Dawson Engler for technical discussions about this work and other applications of these techniques. Thanks to Riad S. Wahby for uploading our huge artifact (and more!), and to Ranjit Jhala for early discussions. Finally, thanks to Chris Fallin, Nick Fitzgerald, Dan Gohman, Jan de Mooij, Nicolas B. Pierron, and Luke Wagner at Firefox; we can't imagine a more thoughtful and responsive group of people. This work was supported in part by gifts from Cisco, Fastly, Google, Mozilla, and Qualcomm; by the NSF under Grant Number CCF-1918573; and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] 18.23 FJCVTZS. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0801g/hko1477562192868.html.
[2] Bug 1027510. https://bugzilla.mozilla.org/show_bug.cgi?id=1027510.
[3] Bug 1493900. https://bugs.webkit.org/show_bug.cgi?id=1493900.
[4] Bug 1595329. https://bugzilla.mozilla.org/show_bug.cgi?id=1595329.
[5] Bug 185694. https://bugs.webkit.org/show_bug.cgi?id=185694.
[6] Bug 765127. https://bugzilla.mozilla.org/show_bug.cgi?id=765127.
[7] Bug 765128. https://bugzilla.mozilla.org/show_bug.cgi?id=765128.
[8] Bug 943303. https://bugzilla.mozilla.org/show_bug.cgi?id=943303.
[9] Bug 950438. https://bugzilla.mozilla.org/show_bug.cgi?id=950438.
[10] Bug 950438. https://bugzilla.mozilla.org/show_bug.cgi?id=950438.
[11] googleprojectzero/fuzzilli. https://github.com/googleprojectzero/fuzzilli.
[12] Issue 1390. https://bugs.chromium.org/p/project-zero/issues/detail?id=1390.
[13] Issue 1396. https://bugs.chromium.org/p/project-zero/issues/detail?id=1396.
[14] Issue 1530. https://bugs.chromium.org/p/project-zero/issues/detail?id=1530.

[15] Issue 1544386. https://bugzilla.mozilla.org/show_bug.cgi?id=1544386.
[16] Issue 1669. https://bugs.chromium.org/p/project-zero/issues/detail?id=1699.
[17] Issue 1775. https://bugs.chromium.org/p/project-zero/issues/detail?id=1775.
[18] Issue 1791. https://bugs.chromium.org/p/project-zero/issues/detail?id=1791.
[19] Issue 1809. https://bugs.chromium.org/p/project-zero/issues/detail?id=1809.
[20] Issue 1810. https://bugs.chromium.org/p/project-zero/issues/detail?id=1810.
[21] Issue 1876. https://bugs.chromium.org/p/project-zero/issues/detail?id=1876.
[22] Issue 762874. https://bugs.chromium.org/p/chromium/issues/detail?id=762874.
[23] Issue 880207. https://bugs.chromium.org/p/chromium/issues/detail?id=880207.
[24] Klee and C++ Analysis Target. https://github.com/klee/klee/issues/852.
[25] Promote Memory to Register. https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register.
[26] Range::ursh. https://searchfox.org/mozilla-central/source/js/src/jit/RangeAnalysis.cpp#1026.
[27] Security/CryptoEngineering/HACL*. https://wiki.mozilla.org/Security/CryptoEngineering/HACL*.
[28] Testengineering/performance/Talos/tests. https://wiki.mozilla.org/TestEngineering/Performance/Talos/Tests.
[29] The C++ Standard. https://isocpp.org/std/the-standard.
[30] z3: Bindings for the Z3 Theorem Prover. https://hackage.haskell.org/package/z3.
[31] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. Language-Independent Sandboxing of Just-in-Time Compilation and Self-Modifying Code. In *PLDI*, 2011.
[32] Apple. JetStream 2. https://browserbench.org/JetStream/.
[33] Apple. SunSpider 1.0.2 JavaScript Benchmark. https://webkit.org/perf/sunspider-1.0.2/sunspider-1.0.2/driver.html.
[34] S. Barati. Spread's Effects are Modeled Incorrectly Both in AI and in Clobberize. https://bugs.webkit.org/show_bug.cgi?id=181867.
[35] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. SoK: Computer-Aided Cryptography. Cryptology ePrint Archive, Report 2019/1393, 2019. https://eprint.iacr.org/2019/1393.
[36] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.smt-lib.org.
[37] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *Workshop on Satisfiability Modulo Theories*, 2010.
[38] A. Barrière, S. Blazy, and D. Pichardie. Towards Formally Verified Just-in-Time Compilation. In *CoqPL*, 2020.
[39] H. Becker, E. Darulova, M. O. Myreen, and Z. Tatlock. Icing: Supporting Fast-Math Style Optimizations in a Verified Compiler. In *CAV*, 2019.
[40] H. Becker, P. Panchekha, E. Darulova, and Z. Tatlock. Combining Tools for Optimization and Analysis of Floating-Point Computations. In *International Symposium on Formal Methods*, 2018.
[41] I. Beer. A Very Deep Dive into iOS Exploit Chains Found in the Wild. https://googleprojectzero.blogspot.com/2019/08/a-very-deep-dive-into-ios-exploit.html, Aug. 2019.
[42] K. Bhargavan, C. Fournet, and M. Kohlweiss. mitls: Verifying Protocol Implementations Against Real-World Attacks. *IEEE Security & Privacy*, 2016.
[43] A. Biondo. Exploiting the Math.expm1 typing bug in V8. https://abiondo.me/2019/01/02/exploiting-math-expm1-v8, Jan. 2019.
[44] D. Blazakis. Interpreter exploitation. In *WOOT*, Aug. 2010.
[45] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A Trusted Mechanised JavaScript Specification. In *POPL*, 2014.
[46] M. Bodin and A. Schmitt. A Certified JavaScript Interpreter. In *JFLA*, 2013.
[47] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *Symposium on Computer Arithmetic*, 2013.
[48] F. Brown, J. Renner, A. Nötzli, S. Lerner, H. Shacham, and D. Stefan. Towards a Verified Range Analysis for JavaScript JITs: Extended Version. https://vera-extended.programming.systems, 2020.
[49] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.
[50] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *SOSP*, 2017.
[51] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *SOSP*, 2015.
[52] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool For Random Testing of Haskell Programs. In *ICFP*, 2000.
[53] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, 1977.
[54] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian. Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *TACAS*, 2018.
[55] D. Delmas and J. Souyris. Astrée: from Research to Industry. In *International Static Analysis Symposium*, 2007.
[56] K. Dewey, J. Roesch, and B. Hardekopf. Fuzzing the Rust Typechecker using CLP. In *ASE*, 2015.
[57] E. ECMAScript, E. C. M. Association, et al. Ecmascript language specification, 2011.
[58] J. Fetiveau. Attacking TurboFan. TyphoonCon, June 2019. https://doar-e.github.io/presentations/typhooncon2019/AttackingTurboFan_TyphoonCon_2019.pdf.
[59] J. Fetiveau. Circumventing Chrome's Hardening of Typer Bugs. https://doar-e.github.io/blog/2019/05/09/circumventing-chromes-hardening-of-typer-bugs/, May 2019.
[60] J. Fetiveau. Introduction to TurboFan. https://doar-e.github.io/blog/2019/01/28/introduction-to-turbofan/, Jan. 2019.
[61] P. Gardner, S. Maffeis, and G. Smith. Towards a Program Logic for JavaScript. In *POPL*, 2012.
[62] P. Gardner, G. Smith, C. Watt, and T. Wood. A Trusted Mechanised Specification of JavaScript: One Year On. In *CAV*, 2015.
[63] S. Groß. The Art of Exploitation: Attacking JavaScript Engines: A Case Study of JavaScriptCore and CVE-2016-4622. *Phrack*, Oct. 2016. http://phrack.org/papers/attacking_javascript_engines.html.
[64] S. Groß. Attacking Client-Side JIT Compilers. Black Hat, Aug. 2018. https://saelo.github.io/presentations/blackhat_us_18_attacking_client_side_jit_compilers.pdf.
[65] S. Groß. FuzzIL: Coverage guided fuzzing for JavaScript engines. Master's thesis, Karlsruhe Institute of Technology, 2018. https://saelo.github.io/papers/thesis.pdf.
[66] S. Groß. Fuzzilli: (Guided)-Fuzzing for JavaScript Engines. OffensiveCon 2019, Feb. 2019. https://saelo.github.io/presentations/offensivecon_19_fuzzilli.pdf.
[67] S. Groß. JIT Exploitation Tricks. 0x41Con 2019, May 2019. https://saelo.github.io/presentations/41con_19_jit_exploitation_tricks.pdf.
[68] S. Groß. JSC Exploits. https://googleprojectzero.blogspot.com/2019/08/jsc-exploits.html, Aug. 2019.
[69] S. Groß. The Art of Exploitation: Compile Your Own Type Confusions: Exploiting Logic Bugs in JavaScript JIT Engines. *Phrack*, May 2019.

http://phrack.org/papers/jit_exploitation.html.
[70] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. In *ECOOP*, 2010.
[71] S. Guo and J. Palsberg. The Essence of Compiling with Traces. In *POPL*, 2011.
[72] S. Guo and J. Palsberg. The Essence of Compiling with Traces. In *POPL*, 2011.
[73] B. Hackett and S. Guo. Fast and Precise Hybrid Type Inference for JavaScript. In *PLDI*, 2012.
[74] H. Han, D. Oh, and S. K. Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *NDSS*, 2019.
[75] D. Jang, Z. Tatlock, and S. Lerner. Establishing Browser Security Guarantees Through Formal Shim Verification. In *USENIX Security*, 2012.
[76] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A Formally-Verified C Static Analyzer. In *POPL*, 2015.
[77] B. Keith. Attacking Edge through the JavaScript Compiler. OffensiveCon 2019, Feb. 2019. https://github.com/bkth/Attacking-Edge-Through-the-JavaScript-Compiler.
[78] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal Verification of an OS Kernel. In *SOSP*, 2009.
[79] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A Verified Implementation of ML. In *POPL*, 2014.
[80] S. Kundu, Z. Tatlock, and S. Lerner. Proving Optimizations Correct Using Parameterized Program Equivalence. In *PLDI*, 2009.
[81] S. Lerner, T. Millstein, and C. Chambers. Automatically Proving the Correctness of Compiler Optimizations. In *PLDI*, 2003.
[82] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In *POPL*, 2005.
[83] X. Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *POPL*, 2006.
[84] W. Lian, H. Shacham, and S. Savage. Too LeJIT to Quit: Extending JIT Spraying to ARM. In *NDSS*, 2015.
[85] W. Lian, H. Shacham, and S. Savage. A Call to ARMs: Understanding the Costs and Benefits of JIT Spraying Mitigations. In *NDSS*, 2017.
[86] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably Correct Peephole Optimizations with Alive. In *PLDI*, 2015.
[87] S. Maffeis, J. C. Mitchell, and A. Taly. An Operational Semantics for JavaScript. In *APLAS*, 2008.
[88] P. Martin. Responding to Firefox 0-days in the Wild. https://blog.coinbase.com/responding-to-firefox-0-days-in-the-wild-d9c85a57f15b, Aug. 2019.
[89] D. Menendez, S. Nagarakatte, and A. Gupta. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *International Static Analysis Symposium*, 2016.
[90] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. Verified Peephole Optimizations for CompCert. In *PLDI*, 2016.
[91] M. O. Myreen. Verified Just-In-Time Compiler on x86. In *POPL*, 2010.
[92] G. C. Necula. Translation Validation for an Optimizing Compiler. In *PLDI*, 2000.
[93] B. Niu and G. Tan. RockJIT: Securing Just-in-Time Compilation using Modular Control-Flow Integrity. In *CCS*, 2014.
[94] A. Nötzli and F. Brown. LifeJacket: Verifying Precise Floating-Point Optimizations in LLVM. In *SOAP*, 2016.
[95] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically Improving Accuracy for Floating Point Expressions. In *PLDI*, 2015.
[96] D. Park, A. Stefănescu, and G. Roşu. KJS: A Complete Formal Semantics of JavaScript. In *PLDI*, 2015.
[97] T. Park, K. Dhondt, D. Gens, Y. Na, S. Volckaert, and M. Franz. NOJITSU: Locking Down JavaScript Engines. 2020.
[98] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *TACAS*, 1998.
[99] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *DLS*, 2012.
[100] Z. Rakamarić and M. Emmi. SMACK: Decoupling Source Language Details from Verifier Implementations. In *CAV*, 2014.
[101] V. S. Rao. Writeup for CVE-2019-11707. https://blog.bi0s.in/2019/08/18/Pwn/Browser-Exploitation/cve-2019-11707-writeup/, Aug. 2019.
[102] C. Reis, A. Moshchuk, and N. Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security*, 2019.
[103] C. Rohlf and Y. Ivnitskiy. Attacking Clientside JIT Compilers. Black Hat, Aug. 2011. https://media.blackhat.com/bh-us-11/Rohlf/BH_US_11_RohlfIvnitskiy_Attacking_Client_Side_JIT_Compilers_WP.pdf.
[104] S. Röttger. A Guided Tour through Chrome's JavaScript Compiler. Zer0Con, Apr. 2019. https://docs.google.com/presentation/d/1DJcWByz11jLoQyNhmOvkZSrkgcVhlIIlCHmal1tGzaw.
[105] S. Röttger. Trashing the Flow of Data. https://googleprojectzero.blogspot.com/2019/05/trashing-flow-of-data.html, May 2019.
[106] H. Samet. *Automatically Proving the Correctness of Translations Involving Optimized Code.* PhD thesis, 1975.
[107] B. Shingarov. Formal Verification of JIT by Symbolic Execution. In *VMIL*, 2019.
[108] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button Verification of File Systems via Crash Refinement. In *OSDI*, 2016.
[109] A. Sintsov. JIT-Spray Attacks & Advanced Shellcode. *HITBSecConf*, 2010.
[110] A. Souchet. A Journey into IonMonkey: Root-Causing CVE-2019-9810. https://doar-e.github.io/blog/2019/06/17/a-journey-into-ionmonkey-root-causing-cve-2019-9810/, June 2019.
[111] V. St-Amour and S. Guo. Optimization Coaching for JavaScript. In *ECOOP*, 2015.
[112] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *POPL*, 2015.
[113] J. Taneja, Z. Liu, and J. Regehr. Testing Static Analyses for Precision and Soundness. In *CGO*, 2020.
[114] Z. Tatlock and S. Lerner. Bringing Extensibility to Verified Compilers. In *PLDI*, 2010.
[115] L. Todesco. A Few JSC Tales. Objective by the Sea, June 2019. http://iokit.racing/jsctales.pdf.
[116] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *Journal of the ACM.*
[117] N. Wang. V8 exploit. http://eternalsakura13.com/2018/05/06/v8/, May 2018.
[118] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *OSDI*, 2014.
[119] A. Wirfs-Brock. ECMAScript 2015 Language Specification – Math.expm1(x). https://www.ecma-international.org/ecma-262/6.0/#sec-math.expm1, 2015.
[120] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *PLDI*, 2011.
[121] M. V. Yason. Understanding the Attack Surface and Attack Resilience of Project Spartan's (Edge) New EdgeHTML Rendering Engine. Black Hat, Aug. 2015. https://www.blackhat.com/docs/us-15/materials/us-15-Yason-Understanding-The-Attack-Surface-And-Attack-Resilience-Of-Project-Spartans-New-EdgeHTML-Rendering-Engine-wp.pdf.
[122] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *POPL*, 2012.
[123] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal Verification of SSA-Based Optimizations for LLVM. In *PLDI*, 2013.

[124] Q. Zhao. Story1 Mom What Is Zero Multiplied By Infinity. https://blogs.projectmoon.pw/2019/01/13/Story1-Mom-What-Is-Zero-Multiplied-By-Infinity, Jan. 2019.

[125] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *CCS*, 2017.