Icarus: Trustworthy Just-In-Time Compilers with Symbolic Meta-Execution

Naomi Smith
UCSD

Abhishek Sharma*
UT Austin

John Renner
UCSD

David Thien
UCSD

Fraser Brown
CMU

Hovav Shacham
UT Austin

Ranjit Jhala
UCSD

Deian Stefan
UCSD

Abstract

Just-in-time (JIT) compilers make JavaScript run efficiently by replacing slow JavaScript interpreter code with fast machine code. However, this efficiency comes at a cost: bugs in JIT compilers can completely subvert all language-based (memory) safety guarantees, and thereby introduce catastrophic exploitable vulnerabilities. We present Icarus: a new framework for implementing JIT compilers that are automatically, formally verified to be safe, and which can then be converted to C++ that can be linked into browser runtimes. Crucially, we show how to build a JIT with Icarus such that verifying the JIT implementation statically ensures the security of *all* possible programs that the JIT could ever generate at run-time, via a novel technique called *symbolic meta-execution* that encodes the behaviors of all possible JIT-generated programs as a single Boogie *meta-program* which can be efficiently verified by SMT solvers. We evaluate Icarus by using it to re-implement components of Firefox's JavaScript JIT. We show that Icarus can scale up to expressing complex JITs, quickly detects real-world JIT bugs and verifies fixed versions, and yields C++ code that is as fast as hand-written code.

1 Introduction

The modern web runs on JavaScript, an extremely dynamic language where even basic operations would execute slowly without tremendous optimization work behind the scenes. On paper, a simple task like checking the length of an array (`array.length`) requires the JavaScript engine to find what the property `.length` refers to with an expensive run-time search, and then chase pointers and resolve dynamic types to read the actual length. In practice, modern browsers run JavaScript *quickly*. This is because browsers use *inline caches* (*ICs*) to optimize away such expensive computations via *just-in-time* (*JIT*) compilation [8, 11, 12, 17]. With ICs, the JavaScript engine tracks the types of the inputs to expressions like `array.length` and what operations were performed to compute the result, and then generates specialized machine-code *fast path* snippets—or "IC stubs"—that efficiently handle specific cases that were previously encountered. So, after the first (slow) execution of `array.length`, subsequent executions can skip the slow detour into the engine and directly run the fast path instead.

Unfortunately, this speed comes at the cost of security. The IC stubs produced by the JIT are fast because they replace the slow JavaScript interpreter with raw, low-level machine operations that, e.g., directly access memory. These operations may be safe to run in the original context which triggered the generation of the IC stub; however, once linked into the rest of the JavaScript program, the IC stub becomes exposed to any *future* inputs supplied by the user (e.g., any future values of `array` once a stub has been generated for `array.length`). JIT authors put in run-time *guards* that determine when the IC stub is safe to run, but bugs in the JIT can cause necessary guards to be omitted, allowing the stub to execute in inappropriate contexts and thereby completely subvert all language-based safety guarantees. This, in turn, allows attackers to compromise the security of the JavaScript engine and the host browser.

JavaScript engines are not just theoretically vulnerable: they are actively exploited. Of the 19 browser bugs known to have been exploited in-the-wild in 2023, 9 were JavaScript engine bugs [41]. Both Firefox and Chrome fell to JavaScript engine bugs at the Pwn2Own contest in March 2024. And all seven high-severity security bugs fixed in April's Firefox 125 release were in the JavaScript engine—two of them in the IC implementation. Existing tools like fuzzers struggle to find subtle IC bugs even when specifically tuned for the problem, because the circumstances necessary to trigger them and then cause a crash are so specific [32]. Worse, reasoning carefully about whether a fix is actually secure is notoriously hard, especially within a reasonable time frame for responding to security vulnerabilities in products relied on by billions. Upon discovering such issues, browser teams are forced to sacrifice performance by entirely disabling large categories of optimizations [15, 18, 32].

*Work done while at UCSD.

SOSP '24, November 4–6, 2024, Austin, TX, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695949

Writing and testing JITs is particularly hard because the unsafe behavior manifests not in code explicitly written by the JIT developers, but in stub code that is *dynamically generated* at runtime—and JIT compilers change the code they generate based on heuristics tracked during program execution. This means unsafe IC stub code may only be generated under particular circumstances and after multiple phases of re-optimization. The principled approach to getting this code right is through formal verification. However, previous JIT verification efforts (e.g., for eBPF [35, 44]) have focused on generating low-level, straight-line assembly, which don't translate to the more complex browser setting. To establish the safety of the JavaScript JIT, complex control-flow operating on rich data-types must be precisely tracked through multiple layers of code generation, the execution of generated code, and interactions with the higher-level JavaScript language runtime.

In this paper, we bridge this gap and reconcile speed and security via Icarus: a new framework for implementing JIT compilers that produce efficient IC stubs that are automatically and formally verified to be safe. The key challenge we face, when comparing to traditional verification which checks the safety of a single program, is that verifying a JavaScript JIT requires verifying the safety of *all possible IC stubs that a particular JIT could ever generate at run-time.* We solve this challenge via four concrete contributions.

1. Symbolic Meta-Execution §2. Our first contribution is a new technique for verifying IC stub generators by converting them into a *meta-stub*: a single program whose possible execution traces encompass the possible execution traces of *all* IC stubs that could be generated by the JIT. Our meta-stubs have two phases: (1) a *generator* phase that simulates the JIT's stub generator and any intermediate compilation steps, and collects the final generated sequence of low-level instructions in an *instruction buffer*; and (2) an *interpreter* phase where the meta-stub loops over the instruction buffer, interpreting each instruction as it goes. We can verify the safety of the meta-stub by symbolically executing it, i.e., by symbolically executing the stub generator *and* the interpreter that loops over the instructions produced by the generator. If verification passes, then by construction, this implies the safety of all possible stubs generated by the JIT.

2. Optimization for Practical Symbolic Meta-Execution. Our second contribution is an optimization which makes symbolic meta-execution practical. A naive implementation of symbolic meta-execution suffers from path explosion: off-the-shelf symbolic executors get stuck in the interpreter phase, trying every permutation of instructions that could fit in the instruction buffer. Looping over an instruction buffer that holds n symbolic instructions, each of which can be one of k instructions, causes the executor to explore roughly k^n paths to determine whether each is either safe or infeasible. In practice, most of these paths are infeasible—the code stub generator would never actually output most sequence of instructions. We distill this insight by introducing a domain-specific optimization: Icarus statically tracks instruction *emits* and control-flow *labels* generated in the JIT to build a *control flow automaton* (*CFA*) describing the space of possible control-flow transfers between instructions across all programs that may be output by the generator phase. We then use this automaton to explicitly constrain symbolic execution of the interpreter loop to only those paths corresponding to the very sparse set of feasible instruction sequences. This is the fundamental to making symbolic meta-execution scale to real-world JITs—and ensure Icarus does not wither away exploring infeasible paths.

3. Framework for IC Generation §3. Our third contribution is the Icarus domain-specific language and framework for building verified JITs. In Icarus, the programmer specifies the JIT platform by describing (1) the *ops* that form the higher-level atomic building-blocks of generated IC stubs, (2) target low-level ops to which the source-level ops are compiled, (3) any intermediate *compiler*s from source ops to target ops, and (4) an *interpreter* that encodes the semantics and safety requirements of the low-level target ops and the JIT runtime. Using this platform, the JIT developer can write individual IC stub generators that translate operations in the source language (in our case, JavaScript) into stub code. The Icarus framework then (1) implements symbolic meta-execution by automatically converting the IC stub generators along with the JIT platform into optimized meta-stubs that can be verified via symbolic execution; and (2) translates the IC stub generators into executable, production-ready C++ that can be integrated into a host application like a web browser.

4. SpiderMonkey implementation and evaluation §4. We implement Icarus in roughly 20K SLOC of Rust and use Icarus to reimplement a significant part of Firefox's inline cache generator. Specifically, we port and verify 21 high-level IC stub generators, which output ops in SpiderMonkey's *CacheIR* intermediate representation. SpiderMonkey compiles CacheIR ops down to low-level *MacroAssembler* (*MASM*) instructions; we port and verify the subset of the CacheIR compiler in SpiderMonkey's "Ion" JIT tier that covers all ops which may be output by these stub generators, and contribute an executable semantics for (a subset of) MacroAssembler in the form of an interpreter implemented in Icarus. We find that Icarus can be used to implement complex JIT programs and high-level JIT security invariants, quickly detect real-world JIT bugs (<30 seconds) and verify fixed versions (in under a minute), and yield C++ code which, when integrated back into Firefox, passes all regression tests with no performance overhead. To our knowledge, this is the first effort to verify real-world JavaScript JIT inline caches—and the first effort to even define a formal (and executable) semantics for core browser IRs like CacheIR and MASM.

```
.Input $0, Value
.Output Value

GuardToObject $0 Ⓐ

# In specialized mode only:
GuardShape $0, «addr of shape» Ⓑ
# In megamorphic mode only:
GuardHasGetterSetter $0, "length", «addr of get/set» Ⓒ

LoadTypedArrayLenInt32Result $0 Ⓓ
ReturnFromIC
```

Figure 1. Two forms of CacheIR IC stubs for handling `TypedArray .length` reads generated by SpiderMonkey. In "specialized" mode, SpiderMonkey generated a `GuardShape` instruction; in "megamorphic" mode, it used a `GuardHasGetterSetter` instruction instead.

2 Overview

We explain the importance of inline caches (ICs) (§2.1), why writing secure ICs is difficult (§2.2), and how the Icarus domain-specific language lets us implement verified ICs via symbolic meta-execution (§2.3).

2.1 A Primer on Inline Caching

JS interpreters are slow. Consider the following JavaScript code snippet that repeatedly evaluates `array.length`:

```
for (let i = 0; i < array.length; i++) { /* ... */ }
```

JS semantics make each `.length` access extremely slow, requiring that the runtime resolve the type of `array` and the `.length` property, and then choose between (1) reading `length`'s value from a field inlined into the `array` object's memory, (2) reading from separately-allocated memory linked to `array` through a pointer, (3) executing arbitrary user-supplied (getter) code to compute the value of `length` on-demand, or (4) performing some even more exotic operation to return the value of the `length` property. Repeating the above at *every* iteration would slow the web to a crawl.

Inline Caches Make JS Fast. Modern JS engines track the types of the inputs to an expression like `array.length` and what operations are ultimately performed to compute the result, and then generate a specialized machine-code snippet or *inline cache (IC) stub* that handles the specific case that was encountered in the first iteration. Thus, after interpreting `array.length` via the "slow path" the first time around, on subsequent iterations, the engine can use the fast IC stub, which makes the overall loop run efficiently.

Generating IC Stubs Using CacheIR. Firefox's JS engine *SpiderMonkey* generates IC stubs in a special-purpose language called *CacheIR*. The actual CacheIR stubs SpiderMonkey generates at runtime can depend upon various heuristic modes that govern how reusable the stubs should be. For example, CacheIR stubs can be generated in "specialized" mode (the default), which is meant to efficiently handle only the exact types that triggered stub generation; or in "megamorphic" mode, where the generated stub handles a wider range of inputs, but at a small performance cost.

CacheIR Stubs for `array.length`. Figure 1 shows two different CacheIR stubs generated by SpiderMonkey for handling `array.length` when `array` is a *`TypedArray`*: a densely-packed collection of homogenously-typed values. Both stubs begin by using `GuardToObject` to check that the input `array` (represented as `$0`) is an object (Ⓐ). In specialized mode, the generated stub follows this with a `GuardShape` instruction (Ⓑ), which confirms that `array` is in fact a `TypedArray` by comparing its shape (i.e., dynamic type) to that of the original input `array` value that prompted the generation of the stub. In megamorphic mode, `GuardShape` is replaced with the slower but more general `GuardHasGetterSetter` instruction (Ⓒ), which checks that the input object has a `.length` property that resolves to the built-in `length` getter from the `TypedArray` class. After the guards, the CacheIR stub implements the optimized "fast path" for the actual `.length` computation: the `LoadTypedArrayLenInt32Result` instruction (Ⓓ) reads the `.length` property directly from a field in `array`'s memory and stores it in the stub's output register as a boxed 32-bit integer. Once the stub is generated, compiled to machine code, and linked into the process, SpiderMonkey splices in a jump to the start of the IC stub at the start of the slow path for the expression `array.length` (hence the name "inline" cache). If any of the guards in the stub fail, control returns to the slow path, and a new stub is generated to specialize the unhandled case for future executions.

2.2 Inline Cache Bugs Compromise Security

SpiderMonkey produces fast stub code by compiling the CacheIR ops of generated IC stubs to a lower-level instruction set, *MacroAssembler* (*MASM*), an abstraction over platform-specific assembly which contains potentially-dangerous primitives like direct memory accesses. In particular, the generated MASM in the stub's "fast path" might have been safe to run with the original inputs that prompted stub generation, but once linked into the JS script, the stub is exposed to *future* inputs supplied by the user (e.g., future values of `array` once a stub has been generated for `array.length`).

Guards Enforce Safety. The *guards* at the start of the IC stubs check that the low-level MASM that follows them is safe to run, or else bail execution out to the slow path. If the guards do not suffice to ensure the safety of the fast path, an attacker can exploit this mismatch to compromise the security of the JavaScript engine. Consider the "specialized" variant of the `TypedArray .length` stub from Figure 1. Omitting either of the `GuardToObject` or `GuardShape` instructions would result in an exploitable type confusion vulnerability. This is because the CacheIR `LoadTypedArrayLenInt32Result` instruction compiles to a raw register-relative memory load

```
AttachDecision
GetPropIRGenerator::tryAttachTypedArrayLength(
  HandleValue value, ValueId valueId,
  HandlePropertyKey key
) {
  Object object = /* get from value */;
  // Check input is an object.
  ObjectId objectId =
    writer.guardToObject(valueId); Ⓔ
  // Check input is a TypedArray whose .length is
  // the built-in getter function.
  EmitCallGetterResultGuards(object, holder, key,
    propInfo, objectId, mode_); Ⓕ
  // Generate the TypedArray .length fast path.
  writer.loadTypedArrayLenInt32Result(objectId); Ⓖ
  writer.returnFromIC();
  return AttachDecision::Attach;
}
```

```
static void EmitCallGetterResultGuards(
  HandleNativeObject object,
  HandleNativeObject holder, HandlePropertyKey key,
  PropertyInfo propInfo, ObjectId objectId,
  ICMode mode
) {
  if (mode == ICMode::Specialized) {
    writer.guardShape(objectId,
      object->shape()); Ⓗ
    /* ... */
  } else {
    GetterSetter* gs =
      holder->getGetterSetter(propInfo.slot());
    writer.guardHasGetterSetter(objectId, id, gs); Ⓘ
  }
}
```

Figure 2. SpiderMonkey CacheIR stub generator for `TypedArray .length` reads that generates stubs from Figure 1.

in MASM (`LoadPrivateIntPtr`), which is otherwise unprotected from accessing arbitrary memory.

Missing (or Bad) Guards Yield Vulnerabilities. In practice, developers both forget guards and use incorrect guards—and this almost always leads to serious security bugs. Indeed, the "megamorphic" variant of the stub shown in Figure 1—generated by SpiderMonkey from July 2020 to January 2021—contains such a bug. The megamorphic variant uses a `GuardHasGetterSetter` instruction to check only that `array` has a `.length` property that resolves to the expected getter hook from the `TypedArray` class. This check is insufficient to protect the subsequent raw memory load `LoadTypedArrayLenInt32Result` at Ⓓ, because an attacker could create a new JavaScript object `tricky` as follows:

```
const tricky = Object.create(Uint8Array.prototype);
```

This `tricky` object has the memory layout of a plain, empty JavaScript object, but inherits all of the properties and methods of the `TypedArray` subtype `Uint8Array`, including the `length` getter. `tricky` would pass the guards at Ⓐ and Ⓒ in the megamorphic stub, but since `tricky` is smaller than a true `TypedArray`, the register-relative load that `LoadTypedArrayLenInt32Result` at Ⓓ compiles to would read past the end of `tricky`'s memory, interpreting the data there as an integer array length, which an enterprising attacker can turn into an out-of-bounds read and write gadget. This bug was discovered internally at Mozilla [32], and labeled *sec-high*—a rating reserved for "exploitable vulnerabilities which can lead to the widespread compromise of many users requiring no more than normal browsing actions" [31].

IC Stubs are Hard to Get Right. Despite their crucial role, there is no mechanism to help developers ensure that IC stubs in general, and guards in particular, are generated correctly. Indeed, doing so is viciously hard, as plain, hand-rolled C++ is used to both generate CacheIR code and further compile it down to MASM. This makes it difficult for the JIT developer to reason about the data- and control-flow over *all* the possible IC stubs that may be generated at run-time.

SpiderMonkey's `tryAttachTypedArrayLength` method as shown in Figure 2 generated the CacheIR code for both IC stubs from Figure 1. At Ⓖ, the code generator emits the code for the fast path that reads the `TypedArray`'s `length` field directly from its memory. To enforce safety, the code generator prefaces the read with guards that validate the type of the input. First, the generator emits the guard that the value is an object at Ⓔ. Then at Ⓕ, it calls a *helper* function `EmitCallGetterResultGuards`—reused across multiple stub generators—that is meant to check the type of the object to ensure the safety of the fast path.

When the helper, shown in Figure 2, is run in specialized `mode`, it emits the correct `GuardShape` guard at Ⓗ. However, when run in a *non-specialized* mode, it emits a weakened "megamorphic" guard at Ⓘ that just checks that the property being accessed has the expected getter/setter hook attached. The challenge is that the megamorphic guard *is* perfectly safe for the *other* contexts where `EmitCallGetterResultCards` is used, where it lets the generated stub handle a wider range of inputs. Unfortunately, though, this guard is insufficient to protect the memory-read that `tryAttachTypedArrayLength` emits after the guards!

2.3 Finding JIT Bugs with Symbolic Meta-Execution

SpiderMonkey developers note in [32] that their fuzzers struggled to find subtle CacheIR bugs like the one in Figure 2, even when specifically tuned to this problem, as the unsafe stub is only generated under extremely specific circumstances. And this difficulty is not unique to SpiderMonkey. In fact, one of the 9 JS engine security bugs known to have been exploited in the wild in 2023 was an IC bug with a similar mismatch of guards between the "monomorphic" and "polymorphic" variants of a stub generated by Chrome's V8 engine [9, 41]. Thus, a key challenge to formally securing

JITs is to ensure the safety of *all* stubs that can possibly be generated by the JIT compiler.

Symbolic Meta-Execution. We developed the Icarus domain-specific language (DSL) to help developers write JITs that are safe by construction. Icarus solves the problem of verifying all possibly-generated stubs via a novel technique called *symbolic meta-execution (SME)*. Its key insight is to use the definition of the stub generator, to compute a single *meta-stub*: a program whose executions correspond to the execution of *all* possible stubs that the generator may produce at runtime. Then, Icarus symbolically executes this meta-stub, ensuring that all possible paths through the meta-stub are safe. This process verifies the safety of all stubs that could be generated by the JIT compiler.

Turning Stub Generators into Meta-stubs. A stub generator simply emits stub code; turning that generator into a meta-stub requires further *interpreting* all possible generated stubs. The resulting meta-stub, which we implement as a Boogie program [27] and symbolically execute with the SMT-based Corral verifier [26], executes in two phases. First, the generator phase symbolically executes the original stub generator and any intermediate compilation passes—for SpiderMonkey, this is the CacheIR stub generator and the subsequent CacheIR-to-MASM compilation step. This phase collects a (symbolic) buffer of all possible generated target instructions—for SpiderMonkey, these are MASM instructions. Second, the interpreter phase interprets the instructions in the buffer. Finally, after both phases complete, the solver has a symbolic representation of the possible executions of every possibly-generated stub.

To actually check safety properties with Icarus, the JIT developer encodes the security requirements of the JIT as assertions associated with each target instruction and the JIT runtime (§3.3). Thus, if Corral statically confirms that all paths of the meta-stub satisfy their assertions, we can rest assured that all possibly-generated stubs are safe.

The next sections walk through the generation and interpretation phases for the meta-stub corresponding to the CacheIR generator from Figure 2.

Phase 1: Generate. To symbolically execute the meta-stub, we start by almost directly translating the IC stub generator and any intermediate compilation layers from their original form (e.g., Figure 2) into Boogie. The Boogie code in Figure 3 starts by initializing the instruction buffer that will hold the generated stub (Ⓙ), then generates the stub code into the buffer (Ⓚ). Zooming into the call at Ⓚ, the Boogie meta-stub calls directly into the CacheIR-to-MASM compiler to generate the MASM instructions implemening the CacheIR instructions (Ⓜ, Ⓝ, Ⓞ), and appends them to the instruction buffer.[1] Finally, the meta-stub calls into the interpreter phase (Ⓛ).

Phase 2: Interpret. Figure 4 shows the Boogie implementation of the interpreter phase of the meta-stub for the generator from Figure 3. Symbolically interpreting the MASM code is tricky, because stubs have non-trivial control-flow like jumps and loops. Like a normal interpreter, the interpreter code here accounts for these by looping through the instruction buffer and, for each MASM instruction, invokes a function that implements the corresponding instruction. Unlike a normal interpreter (e.g., SpiderMonkey), the symbolic interpreter does not have to—and in our current implementation it does not—execute platform-specific assembly. Instead, it lowers to a semantics of the target instructions, i.e., MASM, defined by the JIT developer; in practice, these semantics are code annotated with contracts that capture security invariants of each instruction. For example, Figure 5 shows Boogie code for the `LoadPrivateIntPtr` MASM instruction's semantics in terms of the SpiderMonkey JIT runtime. The assertion Ⓢ encodes that the memory read in `getFixedSlot` is actually in bounds—exactly the invariant violated by the unsafe behavior in the megamorphic `TypedArray` stub.

Catching bugs via Meta-stub Verification. Symbolically executing this meta-stub is equivalent to executing all possible `TypedArray array.length` stubs that could come out of the CacheIR JIT. Thus, Corral *should* flag our Boogie program—the meta-stub code—as containing the same error identified in the bug report. Specifically, Corral *should* produce a *counterexample* that says that when the CacheIR stub is generated with `GuardHasGetterSetter`, interpreting the `LoadPrivateIntPtr` MASM instruction to read the `array`'s `length` field ultimately violates an assertion in the implementation of `$NativeObject~$getFixedSlot`. In practice, Corral does not.

2.4 Optimizing Meta-Stubs with Control-Flow

We let Corral run for a month on the meta-stub from Figure 3, and it had yet to reach a conclusion (i.e., neither SAT nor UNSAT) by the time we finally killed the process. This is not surprising: symbolically executing an interpreter that loops over a symbolically generated buffer of symbolic instructions is recipe for choking. With, say, MASM instruction stubs up to 25 instructions long, comprised of, say, 10 different types of MASM instructions, there are already 10^{25} potential paths through the interpreter loop. Only a tiny handful of these paths can actually be in the instruction buffer after the preceding generator phase. However, symbolic executors, including Corral, can only find those paths after explicitly searching through the space of *all* possible instruction sequences—and thus wither away exploring dead-ends due to the ensuing combinatorial explosion.

Icarus computes a *control-flow automaton* (CFA) that represents a skeleton of the possible paths, and then uses the CFA to build an *optimized* meta-stub that only interprets target instruction sequences that could have been generated,

[1] Note that this is somewhat different from how the original C++ stub generator operates.

```
procedure {:entrypoint}
$entrypoint($value: $Value, $key: $PropertyKey)
{
  // Set up initial state. Ⓙ
  call $CacheIR~$init();
  call $CacheIRCompiler~$init();
  call $valueId := $CacheIR~$defineInputValueId();

  // Generate and compile the stub. Ⓚ
  $MASM~pc := NilPc();
  call $decision :=
    $GetPropIRGenerator~$tryAttachTypedArrayLength(
      $value, $valueId, $key, NilPc());

  // Interpret the stub. Ⓛ
  if ($decision == $AttachDecision~$Attach()) {
    call $MASMInterpreter~interpret();
  }
}
```

```
procedure
$GetPropIRGenerator~$tryAttachTypedArrayLength(
  $value: $Value, $valueId: $ValueId,
  $key: $PropertyKey, pc: Pc
) returns (ret: $AttachDecision) {
  // Check input is an object
  call $GuardToObject($valueId, ConsPcEmitPath(pc, 0)); Ⓜ
  call $objectId := $OperandId~$toObjectId($valueId);
  // Check input is a TypedArray whose .length is
  // the built-in getter function.
  call $EmitCallGetterResultGuards(
    $object, $holder, $key, $propInfo, $objectId,
    $mode, ConsPcEmitPath(pc, 1));
  // Generate the TypedArray .length fast path.
  call $LoadTypedArrayLenInt32Result($objectId,
    ConsPcEmitPath(pc, 2)); Ⓝ
  call $ReturnFromIC(ConsPcEmitPath(pc, 3)); Ⓞ
  ret := $AttachDecision~$Attach();
}
```

Figure 3. Boogie meta-stub corresponding to the CacheIR stub generator in Figure 2.

```
procedure $MASMInterpreter~interpret() {
  // Start at the first instruction in the stub.
  $MASM~pc := $MASM~nextPc(NilPc());
loop: Ⓟ
  // Branch according to the current instruction.
  op := $MASM~opAt($MASM~pc);
  goto
    interpret~$MASM~$LoadPrivateIntPtr,
    interpret~$MASM~$UnboxNonDouble,
    /* ... */;

interpret~$MASM~$LoadPrivateIntPtr:
  // If current instruction is a LoadPrivateIntPtr:
  assume is#$MASM~$LoadPrivateIntPtr(op);
  call $MASMInterpreter~$LoadPrivateIntPtr( Ⓠ
    $valueReg#$MASM~$LoadPrivateIntPtr(op),
    $dstReg#$MASM~$LoadPrivateIntPtr(op),
    $type#$MASM$$LoadPrivateIntPtr(op));
  goto loop;
interpret~$MASM~$UnboxNonDouble:
  // If current instruction is an UnboxNonDouble:
  assume is#$MASM~$UnboxNonDouble(op);
  call $MASMInterpreter~$UnboxNonDouble( ... ); Ⓡ
  goto loop;

// ... etc for other instruction types
}
```

Figure 4. Interpreter phase for the meta-stub in Figure 3.

```
procedure $MASMInterpreter~$LoadPrivateIntPtr(
    $srcAddr: $Address, $dstReg: $Reg) {
  // Read a boxed IntPtr at the given memory
  // address and unbox it.
  call $baseData := $MASMInterpreter~$getData(
    $base#$Address($srcAddr));
  call $srcData := $RegData~$readData(
    $baseData, $offset#$Address($srcAddr));
  // $RegData~$readData ⇢ $NativeObject~$getFixedSlot
  ...
}

procedure $NativeObject~$getFixedSlot(
  $nativeObject: $NativeObject, $slot: $UInt32
) returns (ret: $Value) {
  $shape := $Object~shape($nativeObject);
  call $numFixedSlots := $Shape~numFixedSlots($shape);
  assert $UInt32~lt($slot, $numFixedSlots); Ⓢ
  ...
}
```

Figure 5. Interpreter for MASM `LoadPrivateIntPtr`.

thus tiptoeing around path explosion to efficiently verify JITs.

Computing CFAs. A CFA is an over-approximation of all possible paths through the interpreter when it interprets an arbitrary program from the stub generator—a *much smaller* over-approximation than the set of all possible paths. As an example, consider how Icarus builds the CFA for the `TypedArray` stub generator. Icarus statically follows the paths through the generator to detect that it always first generates the CacheIR operation `GuardToObject`, followed by either `GuardShape` or `GuardHasGetterSetter`, and in either case, ends with `LoadTypedArrayLenInt32Result`; these connections form a directed graph. Icarus then recursively visits the functions that compile these CacheIR instructions to MASM instructions, tracing out similar structures for the MASM instruction sequences that may come out of the compiler, and identifies the possible control-flow transfers between those instructions (including jumps). Finally, Icarus inlines this information into the graph to obtain the CFA.

The left-hand side of Figure 6 shows the CFA, using the larger light-purple boxes to show the structure of the generated CacheIR operations, and the smaller boxes they contain to show the captured control-flow structures of the MASM instructions they may compile to. What's not in the figure: `GuardToUint8Clamped`, `LoadEnclosingEnvironment`, and all the other CacheIR ops—and the many MASM instructions used to implement these ops—that this particular IC stub generator never emits; and, any configurations that can never occur in reality (e.g., like `GuardToObject` coming after `GuardShape`).

Optimizing Meta-stubs. Icarus uses the CFA to produce an optimized meta-stub. This meta-stub constrains the interpreter phase to only execute MASM instruction sequences corresponding to paths in the CFA. Figure 6 shows the optimized version of the interpreter from Figure 4. The optimized interpreter (1) uses `assume` statements to restrict the symbolic execution to only consider specific MASM instructions at each location and (2) uses `goto` instructions to force execution to the next location in the CFA. When Corral verifies the optimized meta-stub, symbolic execution *only* considers the handful of about ten instruction sequences that may be generated by the JIT, and thus finds the counterexample in a speedy 12 seconds. Similarly, the (optimized) meta-stub for the fixed version of the generator successfully verifies in 7 seconds, assuring the JIT developer that the bug has indeed been fixed.

3 The Icarus Framework

In this section we describe the Icarus domain-specific language and framework that lets developers not only implement real JITs using familiar language constructs but also verify their safety using symbolic meta-execution. With Icarus, developers write their JIT as a single codebase. They start by specifying their *JIT platform* comprising:

- the *signatures* of the source-level operations of their IC stubs, and the target-level instructions used to implement those operations efficiently (§3.1);
- the *semantics* of the source-level operations, via functions that *compile* source ops to target instructions (§3.2); and
- the *semantics* of the target instructions, via functions that *interpret* each instruction using a mixture of symbolic contracts and calls into the host application's C++ implementation (§3.3).

Then, they build the top-level IC stub generators of their JIT on top of this platform and use the Icarus toolchain to (1) verify the correctness of the entire stack and (2) extract C++ which they can then integrate into host applications like browsers (§3.4). In the rest of this section we describe these components and their representations in Icarus, with examples from our re-implementations of SpiderMonkey's IC compilers.

3.1 Source Operations and Target Instructions

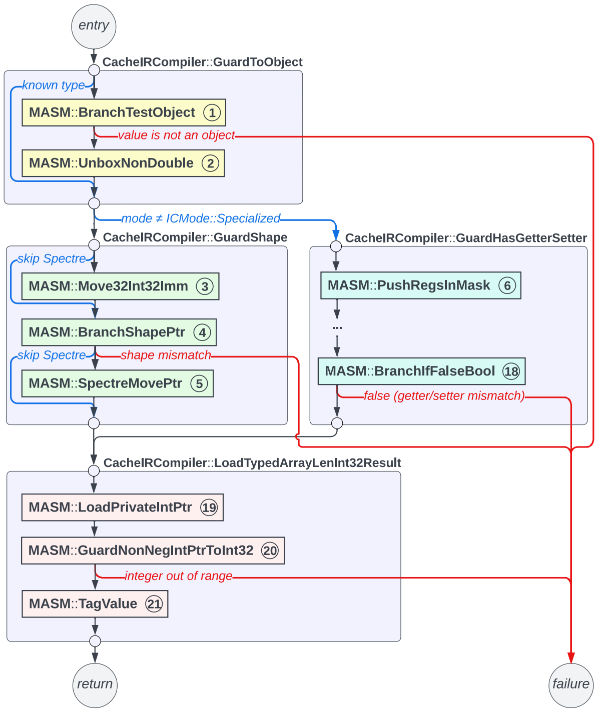
Instruction Sets. To define a particular JIT platform in Icarus, the programmer starts with **`language`** declarations that specify the syntax of source-level operations and target-level instructions (as well as any in-between representations) by listing the constituent "**`op`**s" of each instruction set, together with type signatures describing each **`op`**'s operands. Figure 7 presents extracts from the **`language`** declarations for (source) CacheIR operations and (target) MASM instructions in our SpiderMonkey port, showing a subset of the CacheIR and MASM operations that are relevant to the `TypedArray` bug from §2.2.

Operand Types. The operand types in the **`language`** declarations correspond to types in the existing C++ implementation, which makes it easy to embed Icarus code into the browser's C++ API. For example, operand numbers are represented in the C++ implementation of the CacheIR compiler by typed wrappers like `ValueId`, `ObjectId`, `Int32Id`, etc, descending from a common `OperandId` supertype. Constant fields are represented by typed wrappers around memory offsets into the constant-storage area of the CacheIR stub, with types like `Int32Field`, `GetterSetterField` (for constant pointers to getter/setter pair descriptors), and so on. We declare these as *opaque* types in Icarus, and tie them back to the underlying C++ types when we embed our code into the browser JS engine (§3.4).

3.2 Semantics: Compiling Source Operations

Next, the JIT developer defines a `compiler` from the source-level **`language`** to the target-level **`language`** (e.g., a `CacheIRCompiler` which compiles SpiderMonkey's CacheIR to MASM), and populates the `compiler` definition with callback functions that compile individual source-level **`op`**s to target-level **`op`**s. Figure 8 shows the Icarus code that compiles the `GuardToObject` CacheIR **`op`** into a series of MASM **`op`**s. Icarus's compilation functions look very similar to the original C++ which translates CacheIR operations to MASM. In particular, as with the original C++, Icarus lets the compiler (a) *incrementally* emit the target low-level instructions one-by-one, (b) *condition* instruction emission on values observed at run-time, and (c) generate low-level stub code with *unstructured control flow* using labels and jumps. However, Icarus also engineers the creation and use of labels via built-in language features so that (d) labels can be *statically tracked*, which lets Icarus automatically build the CFAs that enable efficient symbolic meta-execution (§2.4). We detail these next, using `GuardToObject` as the running example.

(a) Code Generation with the `emit` Statement. Existing C++ IC stub generators incrementally append instructions into a buffer containing the target (MASM) program. Icarus follows the same pattern, but uses explicit **`emit`** statements to generate and append target-level instruction to the buffer. At Ⓒ in Figure 8, the compiler emits a MASM guard **`op`** `BranchTestObject` that checks if the boxed JavaScript value in the input register indeed has the type-tag `Object`, and if not, jumps to the supplied failure label. Then at Ⓓ it emits a `UnboxNonDouble` instruction to unbox the value to a raw


entry
CacheIRCompiler::GuardToObject
known type
MASM::BranchTestObject 1
value is not an object
MASM::UnboxNonDouble 2
mode ≠ ICMode::Specialized
CacheIRCompiler::GuardShape
CacheIRCompiler::GuardHasGetterSetter
skip Spectre
MASM::Move32Int32Imm 3
MASM::PushRegsInMask 6
...
MASM::BranchShapePtr 4
skip Spectre
shape mismatch
MASM::BranchIfFalseBool 18
MASM::SpectreMovePtr 5
false (getter/setter mismatch)
CacheIRCompiler::LoadTypedArrayLenInt32Result
MASM::LoadPrivateIntPtr 19
MASM::GuardNonNegIntPtrToInt32 20
integer out of range
MASM::TagValue 21
return
failure

```
procedure $MASMInterpreter~interpret() {
  // Start at the beginning of the stub.
  $MASM~pc := $MASM~nextPc(NilPc());
  // Jump to one of the possible first ops.
  goto
    interpret~$MASM~$BranchTestObject’1,
    interpret~$MASM~$Move32Int32Imm’3,
    interpret~$MASM~$BranchShapePtr’4,
    interpret~$MASM~$PushRegsInMask’6;
interpret~$MASM~$BranchTestObject’1:
  assume emitPath#Pc($MASM~pc) ==
    ConsEmitPath(ConsEmitPath(NilEmitPath(), 0), 0);
  op := $MASM~opAt($MASM~pc);
  call $MASMInterpreter~$BranchTestObject(
    $cond#$MASM~$BranchTestObject(op),
    $reg#$MASM~$BranchTestObject(op),
    $branch#$MASM~$BranchTestObject(op)
  );
  // Jump to one of the possible successor ops.
  goto interpret~$MASM~$UnboxNonDouble’2, exit;
// ... etc for other instruction types
exit:
  // Returned from the stub or bailed out.
  assume
    emitPath#Pc($MASM~pc) == NilEmitPath();
}
```

Figure 6. (L) CFA for meta-stub for generator from Figure 2; (R) CFA-optimized meta-stub interpreter from Figure 4.

```
language CacheIR {
  op GuardToObject(inputId: ValueId);
  op GuardShape(objectId: ObjectId,
    shapeField: ShapeField);
  op GuardHasGetterSetter(objectId: ObjectId,
    idField: IdField, gsField: GetterSetterField);
  op LoadArrayBufferViewLengthInt32Result(
    objectId: ObjectId);
  op ReturnFromIC();
  // ... other CacheIR instructions ...
}

language MASM {
  op BranchTestObject(condition: Condition,
    valueReg: ValueReg, label branch: MASM);
  op UnboxNonDouble(valueReg: ValueReg,
    dstReg: Reg, valueType: JSValueType);
  op LoadPrivateIntPtr(srcAddr: Address,
    dstReg: Reg);
  // ... other MASM ops ...
}
```

Figure 7. Specifying the syntax CacheIR source operations and MASM target instructions in Icarus.

object pointer, which is only safe to execute *if* the preceding guard instruction succeeded.

(b) Conditional Code Generation. At Ⓐ in Figure 8 the compiler starts by testing whether the input operand is already stored unboxed. If so, the conditional turns the entire `GuardToObject` instruction into a no-op by returning without generating any MASM instructions.

```
op GuardToObject(inputId: ValueId) {
  if CacheIRCompiler::knownType(inputId) ==
    JSValueType::Object { return; } Ⓐ
  let inputValueReg =
    CacheIRCompiler::useValueId(inputId);
  let failure = CacheIRCompiler::addFailurePath(); Ⓑ
  emit MASM::BranchTestObject(Condition::NotEqual,
    inputValueReg, failure.label_); Ⓒ
  emit MASM::UnboxNonDouble(inputValueReg,
    ValueReg::scratchReg(inputValueReg),
    JSValueType::Object); Ⓓ
}
```

Figure 8. Icarus code for compiling the `GuardToObject` CacheIR operation to MASM instructions.

(c) Control-Flow with Labels and Jumps. At Ⓑ in Figure 8 the compiler invokes `addFailurePath` to create the `failure` bail-out path that the stub should take to fall back to the JavaScript engine if the stub encounters input it cannot handle. Subsequently-generated MASM operations can then *jump* to the label associated with this failure path. For example, the failure label is passed at Ⓒ to the `BranchTestObject` MASM `op` which performs a test (whether the input is an object) and jumps to the label if the test fails.

(d) Static Label Tracking. JIT compilers like SpiderMonkey create and use labels on-the-fly in order to generate low-level stub code with unstructured control flow. Icarus structures the *declaration*, *use* and *placement* of labels so that we can statically track them to build the CFAs that enable symbolic

```
op CompareInt32Result(jsop: JSOp, lhsId: Int32Id,
    rhsId: Int32Id) {
  // Map the Int32 inputs to physical registers.
  let lhsReg = CacheIRCompiler::useInt32Id(lhsId);
  let rhsReg = CacheIRCompiler::useInt32Id(rhsId);
  // Declare labels that we'll use later. Ⓔ
  label ifTrue: MASM;
  label done: MASM;
  // Compare and conditionally jump to `ifTrue`.
  emit MASM::Branch32(
    Condition::fromJSOp(jsop, true), lhsReg,
    rhsReg, ifTrue); Ⓕ
  // Else, write false and jump to `done`.
  storeConstBool(false,
    CacheIRCompiler::outputReg); Ⓖ
  emit MASM::Jump(done); Ⓗ
  // Bind the `ifTrue` label.
  bind ifTrue; Ⓘ
  storeConstBool(true, CacheIRCompiler::outputReg);
  // Bind the `ifDone` label (to skip true branch).
  bind done; Ⓙ
}
```

Figure 9. Declaring, using and binding labels in Icarus.

meta-execution. Figure 9 shows an example of how labels are used in the Icarus reimplementation of the compiler for CacheIR's `CompareInt32Result` operation that compares two **`Int32`** values. At Ⓔ the compiler *declares* two labels, `ifTrue` and `done`. At this juncture, the labels are "unbound", i.e., not yet placed into the MASM instruction sequence. We can *use* labels before they are bound, e.g., to allow generated code to jump forward to instructions emitted later, to implement if/else constructs. The code emitted at Ⓕ compares the values of `lhsReg` and `rhsReg`, and if the comparison is true, jumps forward to the (as-of-yet unbound) `ifTrue` label. Otherwise at Ⓖ the code writes **`false`** to the output register and jumps to the (also unbound) `done` label, to skip over the `ifTrue` branch. The compiler then uses the **`bind`** statement to place these labels into the instruction sequence by making them refer to the next emitted **`op`** (even if that op is generated across function call boundaries). For example, at Ⓘ, we bind the `ifTrue` label to the MASM op which sets the output register to **`true`**, generated within the `storeConstBool` function call. Finally, at Ⓙ, we actually bind the `done` label, i.e., append the label to the MASM stream to refer to the first MASM instruction that comes next in the stub *after* the code block for the `CompareInt32Result` comparison operation.

Labels are kept distinct from values in that they cannot be stored in variables, returned from functions, or used in expressions (except as arguments to function calls), or—for locally-declared labels—escape the outermost scope in which they are declared. Icarus ensures a locally-declared label *must* be bound exactly once before it goes out of scope, so that there are no jumps to dangling, unbound labels in the final compiled program. Put together, Icarus's invariants ensure that the identity of a label being passed as an argument or operand can be statically determined, which lets Icarus (1) ensure the control-flow is well-formed, and (2) compute the CFAs that enable symbolic meta-execution (§2.4).

```
op BranchTestObject(condition: Condition,
    valueReg: ValueReg, label branch: MASM) {
  assert condition == Condition::Equal ||
    condition == Condition::NotEqual; Ⓚ
  let value =
    MASMInterpreter::getValue(valueReg); Ⓛ
  let valueIsObject = Value::isObject(value); Ⓜ
  if condition == Condition::Equal && Ⓝ
    valueIsObject { goto branch; }
  if condition == Condition::NotEqual &&
    !valueIsObject { goto branch; }
}
op UnboxNonDouble(valueReg: ValueReg,
    objectReg: Reg, valueType: JSValueType) {
  assert valueType != JSValueType::Double;
  // Read the boxed value from the input register.
  let value = MASMInterpreter::getValue(valueReg);
  if valueType == JSValueType::Object {
    let object = Value::toObject(value); Ⓞ
    MASMInterpreter::setObject(dstReg, object);
  } else { /* ... code to unbox other types ... */
  }
}
impl Value {
  refine safe fn toObject(value: Value) -> Object {
    assert Value::isObject(value); Ⓟ
    let obj = unsafe { raw Value::toObject(value) }; Ⓠ
    assume value == raw Value::fromObject(obj); Ⓡ
    obj
  }
}
```

Figure 10. Interpreters for MASM `BranchTestObject` and `UnboxNonDouble` instructions.

3.3 Semantics: Interpreting Target Instructions

The last piece of building a JIT platform in Icarus is defining an `interpreter` for the target-level **`language`**, and populating it with callback functions that specify an *executable semantics* of the target instruction set. Each of these functions interprets a particular target **`op`**, and typically includes a mix of (1) *contracts* specifying key safety and security requirements to be verified by symbolic meta-execution; and (2) *external calls* into the browser's C++ implementation to interact with the JIT runtime and host application (i.e., the browser). This setup gives us a foundation to verify the safety of both the generated code and interactions between generated code and the JIT runtime, and—as a bonus—makes the interpreter executable if translated to C++ by the Icarus toolchain. Figure 10 shows the interpreters for the MASM instructions emitted by the compiler in Figure 8.

`BranchTestObject.` The interpreter function for `BranchTestObject` in Figure 10 starts with an **`assert`**

at Ⓚ that specifies the *precondition* that this **op** must only be called with a valid `condition` operand: `Equal` or `NotEqual` (and not, say, `LessThan`). Next, at Ⓛ, the interpreter calls a helper function `getValue` on `MASMInterpreter` to get the current boxed JavaScript value held in the input `valueReg` at runtime. We then perform the actual test at Ⓜ by calling `Value::isObject`, a C++ function from the SpiderMonkey runtime, on the register `value`. Finally, at Ⓝ, we conditionally **goto** the given `label` depending on whether the test passes and the specific `condition` being tested for, using Icarus's built-in support for labels and jumps (§3.2).

`UnboxNonDouble`. The compiler function for `GuardToObject` in Figure 8 emits an `UnboxNonDouble` instruction only *after* emitting a `BranchTestObject` instruction. The middle of Figure 10 shows the interpreter function for the MASM `UnboxNonDouble` instruction. After some preconditions requiring that the input not be a `Double`, at Ⓞ the interpreter calls the helper function `Value::toObject` to perform the actual unboxing operation if the value is an object. This external function is **refine**d with a contract (at the bottom of Figure 10) which requires the precondition that the input value is indeed an object (Ⓟ). The precondition holds when `UnboxNonDouble` is emitted by the compiler for `GuardToObject` because the preceding `BranchTestObject` **op** explicitly ensures that the input value is an object; if we had omitted this test, the `toObject` function would be unsafe to call, and Icarus's symbolic meta-execution would flag the bug. Fortunately, as we did emit the test, the SME statically verifies that the precondition holds, and hence that it is safe to call the underlying native C++ function `Value::toObject` in the JIT runtime at Ⓠ—the **raw** syntax marks this direct call, bypassing the **refine**d wrapper. Following this raw call is a post-condition that the returned `obj` will indeed be the same one boxed inside `value`, at Ⓡ.

3.4 Building Verified and Executable JITs

With the underlying JIT platform in place, the developer can finally write their top-level IC stub generators that emit **op**s in the source **language** they defined—in our case, CacheIR. Figure 11 shows the Icarus version of the C++ IC stub generator from Figure 2. We designed Icarus so that the ported code has the structure of the original C++, but uses Icarus's domain-specific constructs like the **emit** statement to track the generation of source-level **op**s and, transitively, the target-level **op**s they are compiled to.

Verification. Icarus's **emit** statement and **label** constructs let our toolchain precisely track the possible sequences of CacheIR and MASM instructions generated by the IC stub generator and the control-flows between these instructions, needed to compute the CFA-optimized meta-stub shown in Figure 6. Icarus builds the code-generator phase of the meta-stub by linking the locations where CacheIR ops are **emit**ted in the top-level code generator (Figure 11) with corresponding callback functions in the definition of the CacheIR-to-MASM compiler (§3.2). Icarus then follows the process described in §2.4 to construct the CFA used to constrain symbolic execution of the interpreter phase, tracing connections between **emit**ted MASM ops and statically-tracked **label**s to identify the possible control-flows between MASM ops across the space of possible generated stubs. Each segment of the optimized interpreter loop calls into one of the callback functions defined in the MASM interpreter (§3.3) to interpret a corresponding MacroAssembler op.

The structure described above is translated into Boogie and fed into the Corral verifier for symbolic execution. Given the example buggy `TypedArray .length` IC stub generator from §2, ported into Icarus, the verifier swiftly pinpoints the exact flaw identified by SpiderMonkey developers [32]. Conversely, after applying the developers' fix, we get a meta-stub that Corral successfully verifies, thereby assuring the safety of the IC generator.

Execution. As mentioned above, in addition to translating Icarus code into Boogie for verification, the Icarus toolchain implements another backend which translates it into C++ that can be embedded in the host application, i.e., the browser in our use case. This produces a C++ function for each top-level stub generator, and C++ visitor functions for each of the per-**op** compiler and interpreter callback functions. The developer must connect the external types and functions that they used in their Icarus code with a thin layer of C++ binding code that bridges them to their C++ counterparts. A skeleton for this binding layer is automatically generated by the toolchain. With this binding layer in hand, the developer can wire up their verified and translated JIT code in the host application as a replacement for the original. Organizing the translated code around the visitor pattern makes it straightforward to carry out this swap

```
fn tryAttachTypedArrayLength(
  value: Value, valueId: ValueId,
  key: PropertyKey
) emits CacheIR {
  let object = /* get from value */;
  // Check input is an object.
  emit CacheIR::GuardToObject(valueId);
  let objectId = OperandId::toObjectId(valueId);
  // Check input is a TypedArray whose .length is
  // the built-in getter function.
  emitCallGetterResultGuards(
    object, holder, key, propInfo, objectId,
    GetPropIRGenerator::mode);
  // Generate the TypedArray .length fast path.
  emit CacheIR::LoadArrayLengthInt32Result(objectId);
  emit CacheIR::ReturnFromIC();
  return AttachDecision::Attach;
}
```

Figure 11. Icarus code for the `TypedArray .length` IC stub generator from Figure 2.

piece-by-piece. As more and more pieces of the JIT are incrementally ported into Icarus and verified, the developer gains increasing assurance of their system's safety and security.

4 Evaluation

We implement Icarus as a collection of Rust modules: a shared frontend including parsing, name resolution, and type-checking logic; compilation backends to C++ and to Boogie, as well as the CFA static analyzer; and a separate library for parsing, printing, and optimizing Boogie code (e.g., dead-code elimination). Our implementation totals to 19,804 SLOC of Rust, 858 SLOC of Boogie support code, and 57 SLOC of Icarus prelude code.

To evaluate the Icarus language, verifier, and translation to C++, we port a slice of SpiderMonkey's CacheIR JIT from C++ to Icarus. We answer the following questions:

- Can we implement core JIT IRs in Icarus?
- Is Icarus expressive enough to let developers specify security-relevant invariants about the behavior of JIT-compiled inline cache stubs?
- Can we implement and verify high-level CacheIR code generators?
- Can Icarus catch real-world CacheIR bugs?
- How does the correctness and performance of the C++ code produced by Icarus compare with the original?

All timing and benchmarking described in this section is done on a Lenovo ThinkPad P16 Gen 2 laptop with an Intel Core i9-13980HX processor and 128 GB of RAM, running Ubuntu 22.04.3 LTS virtualized under Windows 11. Numbers quoted are averaged over 10 runs.

4.1 Implementing the CacheIR JIT with Icarus

To answer the first question we use Icarus to implement subsets of the CacheIR and MASM instruction sets and port part of the CacheIR-to-MASM compilation pass. Since our goal is to verify the ported JIT compiler we also use Icarus to give a semantics to MASM by defining an interpreter for MASM and parts of the JavaScript runtime and its associated types and functions. We find that Icarus is expressive enough to implement real-world JIT compilers.

CacheIR compilation. We implement 81 CacheIR instructions of the 334 total. We pick instructions that are common in browser benchmarks (and thus highly tuned), and instruction of different categories (e.g., guards, memory operations, type conversions, external runtime calls). We also include all the CacheIR ops required by the high-level code-generators we port in §4.3–§4.4. For the CacheIR-to-MASM compiler implementation, we hew close to a direct port of the original C++ compiler into Icarus, totalling 1,597 lines of Icarus.

MacroAssembler (and JS runtime) semantics. Since our goal is to verify the CacheIR-to-MASM compiler and the code-generators built on top, we define an interpreter for the 131 different MASM ops that these CacheIR ops can compile to, in 1,891 lines of Icarus. As part of this, we also define the interface to the JavaScript language runtime provided by SpiderMonkey, in 1,135 lines of Icarus. This consists of a mix of stub functions with contracts on the behavior of the underlying C++ function (like `Value::toObject` in §3.3) as well as some SpiderMonkey functions that we port in full. We carefully reverse-engineer the semantics of each MASM op based on its lowering to x86_64 machine code (and interactions with the runtime). This is the first declarative and reusable formal specification of MASM, which until now has been ad-hoc and implicit in the the SpiderMonkey source.

4.2 Invariants about JIT-Compiled CacheIR

To understand if Icarus lets developers verify high-level security properties of high-level code generators, we express those properties as pre- and post-conditions at the lower levels of the JIT and runtime. Icarus lets us encode these invariants as **`assert`** statements, which state properties to be statically checked; and **`assume`** statements, which provide information to the verifier to aid in drawing the right conclusions about our code. These invariants propagate up the layers of the stack to, in our case, ensure that CacheIR and MacroAssembler instructions are used safely. For example, in Figure 10, we **`assert`** that unboxing a `Value` to an `Object` pointer is only permitted if the `Value` has the `Object` tag; this ensures that uses of any MASM instructions which perform this unboxing, like `UnboxNonDouble`, only pass verification if it can be (automatically) proven that they only handle `Values` for which the invariant holds. The invariant may be upheld by the implementations of the MASM instructions themselves; or it may be upheld by the way that the MASM instructions are combined by the CacheIR-to-MASM compiler, like pairing a `UnboxNonDouble` with a preceding `BranchTestObject` to ensure only correctly-tagged `Values` make it through; or it may be punted all the way up to the CacheIR level, becoming a precondition on the CacheIR instructions themselves and putting the onus on the top-level code generator to ensure that those conditions are satisfied.

We find that this approach is effective in verifying many high-level JIT security properties using Icarus, including:

Type Confusion Avoidance. We check that accesses to data in registers, on the stack, and on the heap will all be well-typed at runtime, and that all conversions are well-formed (e.g., boxed `Value` may only be downcast to an `Object` pointer if the verifier can prove that the `Value` is always an `Object`).

Memory Access Bounds-Checks. Raw accesses that MASM makes to memory addresses are validated against the types, bounds, and memory layouts of the base JS runtime types being handled by the interpreter.

Guard Elision Correctness. Potentially-unsafe ops in CacheIR and MASM code must be sufficiently protected by preceding guard ops which rule out invalid cases.

			Time (s)	
Operation	**Code Generator**	**Total LOC**	**Mean**	**σ**
Compare	Any Null/Undef.	778	3.52	0.01
	Int32	707	3.14	0.03
	Strict Diff. Types	564	0.43	0.01
Get Element	Dense Element	1,040	18.05	0.14
	Native Fixed Slot*	1,377	47.86	0.16
Get Property	Args. Object Arg	732	0.98	0.02
	Native Dyn. Slot*	987	4.26	0.03
	Native Fixed Slot*	970	0.86	0.01
	Object Length	1,005	4.10	0.05
Int32 Binary Operator	Add	658	4.32	0.03
	Bitwise	881	46.26	0.39
	Divide	748	16.66	0.13
	Mod	735	13.99	0.04
	Multiply	710	19.87	0.10
	Subtract	669	4.26	0.04
Int32 Unary Operator	Arithmetic	703	2.06	0.01
	Bitwise	645	0.46	0.01
To Property Key	Int32	443	0.18	0.00
	Number (float. pt.)	749	0.44	0.01
	String	444	0.18	0.00
	Symbol	444	0.18	0.00

Figure 12. CacheIR code-generators that we port into Icarus and verify for our evaluation, with verification times. Total Icarus LOC verified is given for each code-generator, including the lines of code in the generator function itself as well as in lower levels of the JIT exercised by the generator and in supporting functions in the call graph. For the code-generators handling native object slots (*), we implement the code paths generating direct accesses in non-megamorphic stubs.

Safe Register Handling. Our port of the CacheIR-to-MASM compiler is checked against a simplified model of the register allocator, asserting that registers are not double-allocated, allocated improperly, or clobbered. We also check the saving and restoring of live-register sets around operations like calls into C++ functions in the runtime.

JavaScript Runtime Call ABI. We model calls into the runtime from generated MASM code and ensure that such calls are (1) well-formed with, e.g., well-typed arguments in their expected locations according to the ABI, and (2) robust against clobbered caller-saved registers.

4.3 Verifying CacheIR Code Generators

We test whether Icarus can successfully *verify* the high-level invariants we describe in the previous section by porting and verifying several CacheIR code-generators from SpiderMonkey's JIT. Figure 12 lists the code-generators we port for this evaluation. Since SpiderMonkey contains 270 code-generators spread across 21 operations, we focus on a subset of operations (e.g., "Get Property") and, within each operation, typically a subset of the code generators. The one exception: we port and verify the entire "To Property Key" operation. We pick different operations to cover different parts of the JavaScript engine and to make use of a broad range of CacheIR and MASM ops under-the-hood. The 21 code-generators that we ultimately ported have a median total LOC of 732 lines of Icarus each, counting lines of code in the top-level generators as well as in the slice of the lower levels of the JIT exercised by each particular generator (computed by summing over their call graphs).

We implement these code-generators on top of the foundation of §4.1; namely, the re-implemented CacheIR-to-MASM compiler, the MASM interpreter, and the JavaScript runtime contracts, which together provide a base executable semantics to verify the CacheIR code-generators against. As described in §2–§3, Icarus takes these layers of code as input and produces a CFA-optimized meta-stub in the form of a verifiable Boogie program for each code-generator. Our verification pipeline runs each meta-stub through the Corral program verifier, which should flag an error if any possible stub program produced by the code-generators could violate one of our invariants given some user input. As Figure 12 shows, Icarus verifies all the code-generators in under a minute (and typically in under four seconds).

		Icarus		No Icarus	
Benchmark	**Unit**	**Mean**	**σ**	**Mean**	**σ**
ARES-6	s ↓	86.69	0.16	87.15	0.33
Octane	score ↑	42,528	402	42,527	404
Six Speed	ms ↓	5,916	50	5,877	46
Sunspider	ms ↓	98	2	97	3
Web Tooling	runs/s ↑	11.79	0.10	11.74	0.11

Figure 13. Results from the five standard JavaScript benchmarks bundled with SpiderMonkey, showing comparable performance between Icarus-enhanced and stock builds of the JavaScript engine. Different benchmarks report their results in different units; ↑ indicates that higher numbers are better, and ↓ that lower is better.

4.4 Reproducing and Catching Real-World Bugs

We next assess Icarus's usefulness in catching a selection of real-world historical CacheIR JIT security bugs from Firefox's bug tracker, covering different engine layers and kinds of bug. For each bug, we implement a corresponding CacheIR code-generator that reproduces the bug (along with any supporting JIT-platform code), and run the buggy code-generator through Icarus's verification pipeline. We then patch the code-generators according to the fixes the SpiderMonkey developers made for each bug, and re-run verification to ensure that it succeeds with the patch applied. In §2–§3 we describe bug 1685925 (incorrectly-optimized `TypedArray .length` accesses [32]) in detail. Figure 14 summarizes our findings: Icarus catches all six bugs in less than thirty seconds each and verifies the fixes in under a minute.

				Verification Time (s)					
	Bug Summary			**Buggy**			**Fixed**		
Bug #	**Occurs During**	**Buggy Layer**	**Violated Invariant**	**Median**	**Mean**	**σ**	**Median**	**Mean**	**σ**
1451976	Truncate Floating Point	CacheIR Compiler	Type Confusion	8.26	8.24	0.06	14.39	14.33	0.14
1471361	Truncate Floating Point	CacheIR Compiler	Stack Consistency	5.36	5.35	0.06	16.75	16.77	0.22
1502143	Get Sparse Element	CacheIR Generator	JS Runtime Invariant	20.59	20.66	0.19	45.20	45.31	0.49
1651732	Get Proxy Element	JS Runtime Function	JS Runtime Invariant	1.06	1.06	0.02	0.81	0.81	0.01
1654947	Int32 Bitwise Shift	CacheIR Compiler	Register Clobbering	1.07	1.07	0.02	0.78	0.79	0.01
1685925	Get TypedArray Length	CacheIR Generator	OOB Memory Read	11.67	11.64	0.09	7.01	7.00	0.02

Figure 14. Previously-reported CacheIR bugs reproduced for our evaluation. **Buggy Layer** refers to the layer of the JIT that introduces the bug. **Verification Time** indicates time taken to catch the bug and to verify the corresponding fix with Icarus.

4.5 Extracting C++ into SpiderMonkey

Icarus is designed with the intention that code written in its DSL can be integrated straightforwardly into a host application. We check how well Icarus measures up to this design goal by integrating our partial re-implementation of the CacheIR compiler back into SpiderMonkey. We evaluate our Icarus-enhanced JavaScript engine against the battery of tools that SpiderMonkey makes available for correctness and performance testing. Our build passes all `jstests` and `jit-test` test suites bundled with SpiderMonkey (over 50,000 test cases). And, as Figure 13 shows, our hardened JIT engine is as fast as the unmodified engine on the five standardized JavaScript benchmark suites that SpiderMonkey includes in its test harness.

5 Discussion and Limitations

Scalability. Like other automated verification tools, we do not expect Icarus's symbolic meta-execution to scale to arbitrary, branch-heavy programs. In our experience, though, careful design and domain-specific optimizations are key to making the verification approach practical. The Icarus DSL, for example, steers developers toward writing programs with statically-analyzable and reducible control-flow using first-class language constructs for code generation (e.g., `emit` and `label`s). This, in turn, makes it possible for us to implement domain-specific optimizations like the CFA (§2.4) and take advantage of solver optimizations. For example, Icarus reifies loops at the source-level and between `emit`ted instructions into source-level loops in the meta-stub program; this lets Corral apply built-in heuristics for loop-verification [26]. Of course, this only goes so far: unbounded loops for which loop invariants cannot be automatically inferred may only be verified up to a tuneable iteration bound, a limitation we inherit from Corral. However, Corral is able to verify all our benchmarks using its automatically inferred invariants.

Specification. Invariants in Icarus are expressed with `assert` and `assume` statements. This means global axioms governing the behavior and usage of data types cannot be directly expressed in Icarus. In practice, in JITs, we have found such global axioms to be expressible as local properties of the code operating on those data types. This, however, does mean developers need to turn potentially high-level, declarative specification into low-level, imperative (and harder to get right) specifications. Extending Icarus with global, declarative specification constructs that automatically generate local assertions, or take advantage of global constructs in Boogie and Corral, could be a fruitful area for future work.

CFAs and verification. While Icarus's *control-flow automaton* (CFA) optimization is "just" an optimization, i.e., it does not introduce false positives or negatives, the verification performance depends on the CFAs Icarus generates. Our CFA-generation algorithm traces the code-generator's control-flow and `emit` statements to generate an *over-approximation* of the sequences of instructions that may be executed at runtime. The over-approximation means that in the worst case, the interpreter phase may explore more paths that necessary, which, in the worst case can limit the scalability of the verification. Since this algorithm is recursive this also means Icarus programs must be *non-recursive.* This may not be a fundamental limitation, but has simplified our implementation.

TCB when verifying JITs with Icarus. Icarus itself has been tested but not verified, so the toolchain and handwritten Boogie support code are both trusted components. When verifying a JIT pipeline, the interpreter at the bottom layer of the stack—in our evaluation, the MASM interpreter—is also trusted: it defines the semantics of the target language. An error in implementing the interpretation of a target-language instruction could lead to incorrectly verifying the JIT pipeline layered on top. Finally, the contracts that developers writes for external functions (not implemented in Icarus) are largely trusted, i.e., they should correctly describe the behaviors of the functions (as relevant to the properties being verified).

Incremental verification with Icarus. Icarus is designed so that JIT code can be implemented or ported to Icarus piecemeal, reaping the benefits of verification without having to migrate an entire application first. Our CacheIR implementation covers a subset of the full complement of CacheIR code generators, CacheIR instruction compilers, and MASM instructions in the interpreter (§4.1), which grew bit-by-bit over the course of our evaluation. When we port additional chunks of the CacheIR system, we start by porting a top-level code generator into Icarus, then port the compilers for any CacheIR instructions the generator may emit which are not yet supported by our implementation, and finally extend the MASM interpreter to cover all MASM instructions which those CacheIR instructions may compile to—so all instructions whose behavior may affect the verification of the ported code generators are included in our model at each step in its evolution. It is straightforward to incorporate external code into verified Icarus programs by encapsulating such code as opaque functions refined with contracts (§3.3). The verifier can be invoked on individual top-level JIT code generators as verification targets, with a custom Boogie dead-code elimination pass (which we make available as a standalone open-source component) cutting the JIT stack down to the minimal vertical slice required for verification. This way, new generators may be added, modified, and verified without having to re-check the entire codebase.

Further advantage could be taken of modular verification to improve verification times for JITs written in Icarus. As discussed in §4.2, some invariants are entirely local to individual instructions, and could be automatically proven by examining the implementations of those instructions on their own (e.g., the type-safety of a register which is allocated, used, and deallocated within the compilation of a single CacheIR instruction to MASM); whereas other invariants can only be proven by examining how instructions are combined (e.g., a `BranchTestObject` MASM instruction making a subsequent `UnboxNonDouble` instruction safe to execute). At present, Icarus always takes a global perspective on the programs it verifies, and so the former, local kind of invariants are repeatedly checked at every location where the corresponding instruction may be emitted. A future iteration of the Icarus verification pipeline could identify these local invariants and check them only once, independently, and then take them as true when checking the program as a whole, thus saving on verification time.

6 Related Work

Compiler and JIT Correctness. There is a rich history of research on proving compiler correctness [28–30, 47]. Some is particular to JITs, including reasoning about correctness of trace-based optimizations [14, 21], proving JIT correctness in a proof assistant [3–5, 33, 44]. In contrast to most of these efforts, which *manually* prove that the JIT's output is semantically equivalent to the source—Barrière, et al. [3] even do this for a JIT with de-optimizations!—Icarus aims for *fully automatic* verification of the security properties of the JIT emitted code. Moreover, we focus on porting and verifying parts of existing, complex JITs (e.g., CacheIR in Firefox) instead of building verified JITs from a clean slate.

Automatic JIT Verification. Prior work has looked at automatically proving correctness of specific *analyses* done by JITs e.g., range analysis [7], as opposed to verifying the correctness of the code emitted by the JIT itself. Previous research has also looked at using SMT-based symbolic execution for push-button JIT verification [34, 35, 39], synthesizing kernel JITs [42], but this is in the context of compiling individual kernel (BPF) JITs where *single* high-level operations are translated into *straight-line blocks* of low-level code, which can be directly verified by (plain) symbolic execution. In contrast, this work focuses on browser JITs and addresses the problem of *unstructured control-flow* in the generated inline caches via meta-execution.

JIT hardening. There is also work on limiting the damage of JIT bugs after they appear, including systems like Shuffler [45] RockJIT [36], JITScope [46] JITGuard [16], JITSec [10], NaClJIT [1], NoJITSu [38], and most recently Chrome's V8 Sandbox [19]. These systems typically use runtime software-based fault isolation or fine-grain access control checks to ensure data or control-flow integrity, even in the face of JIT bugs. Icarus, in contrast, helps developers implement secure JIT and catch vulnerable code at compile time, before the bug can be exploited.

Finding browser JIT bugs. Most browsers have fuzzing teams and run fuzzers: the Firefox team, for example, runs multiple different fuzzers targeting the JavaScript engine [25], while Google runs distributed fuzzers on Chrome on thousands of machines [2]. Every major browser teams fine-tunes fuzzers, including Project Zero's Fuzzilli [20] fuzzer, which uses an IR to explicitly generate programs that exercise parts of the underlying JIT (instead of, say, the JavaScript parser). Many other fuzzers focus on different parts of the JavaScript engine [6, 13, 22–24, 37, 43], going back to jsfunfuzz [40]. Unfortunately, even fuzzers like Fuzzilli [20] and DIE [37], which find deep bugs have and will inevitably miss many JIT bugs; Icarus, in contrast, can help developers find and prove the absence of bugs in cases fuzzers will have an extremely difficult time even making a dent. Of course, running a fuzzer doesn't typically require rewriting parts of the JIT so the need for both approaches is clear.

Acknowledgments

We thank the anonymous reviewers and our shepherd Jon Howell for valuable feedback on our paper submission. We thank Bobby Holley, Jan de Mooij, Iain Ireland, and Yulia Startsev for fruitful discussions and feedback on our work. This material is based upon work supported by the National Science Foundation under Grant Nos. 2120642, 2120696, 2154964, 2155235, and 2327336; and by gifts from Mozilla and the Google V8 team.

References

[1] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. 2011.

Language-Independent Sandboxing of Just-in-Time Compilation and Self-Modifying Code. In *PLDI*. ACM.

[2] Abhishek Arya and Cris Neckar. 2012. Fuzzing for Security. Online: https://blog.chromium.org/2012/04/fuzzing-for-security.html.

[3] Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. 2021. Formally Verified Speculation and Deoptimization in a JIT Compiler. In *POPL*. ACM.

[4] Aurèle Barrière, Sandrine Blazy, and David Pichardie. 2020. Towards Formally Verified Just-in-Time Compilation. In *CoqPL*.

[5] Aurèle Barrière, Sandrine Blazy, and David Pichardie. 2023. Formally Verified Native Code Generation in an Effectful JIT: Turning the CompCert Backend into a Formally Verified JIT Compiler. In *POPL*. ACM.

[6] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-picking: Differential fuzzing of JavaScript engines. In *CCS*. ACM.

[7] Fraser Brown, John Renner, Andres Noetzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Towards a Verified Range Analysis for JavaScript JITs. In *PLDI*. ACM.

[8] Craig Chambers, David Ungar, and Elgin Lee. 1989. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language based on Prototypes. *ACM SIGPLAN Notices* 24, 10 (1989).

[9] Chromium Issue Tracker 2023. Security: [0-day] Bug in the Handling of the Arguments Object. Online: https://issues.chromium.org/issues/40065138.

[10] Willem De Groef, Nick Nikiforakis, Yves Younan, and Frank Piessens. 2010. JITSec: Just-in-time Security for Code Injection Attacks. In *WISSEC*.

[11] Jan de Mooij, Matthew Gaudet, Iain Ireland, Nathan Henderson, and J Nelson Amaral. 2023. CacheIR: The Benefits of a Structured Representation for Inline Caches. In *MPLR*. ACM.

[12] L Peter Deutsch and Allan M Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *POPL*. ACM.

[13] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *NDSS*. Internet Society.

[14] Stefano Dissegna, Francesco Logozzo, and Francesco Ranzato. 2016. An Abstract Interpretation-Based Model of Tracing Just-in-Time Compilation. *ACM TOPLAS* 38, 2 (2016).

[15] Jeremy Fetiveau. 2019. Circumventing Chrome's Hardening of Typer Bugs. Online: https://doar-e.github.io/blog/2019/05/09/circumventing-chromes-hardening-of-typer-bugs/.

[16] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. JITGuard: Hardening Just-in-Time Compilers with SGX. In *CCS*. ACM.

[17] Matthew Gaudet. 2023. An Inline Cache isn't Just a Cache. https://www.mgaudet.ca/technical/2018/6/5/an-inline-cache-isnt-just-a-cache.

[18] Sergei Glazunov. 2021. In-the-Wild Series: Chrome Infinity Bug. Online: https://googleprojectzero.blogspot.com/2021/01/in-wild-series-chrome-infinity-bug.html.

[19] Samuel Groß. 2024. The V8 Sandbox. Online: https://v8.dev/blog/sandbox.

[20] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *NDSS*. Internet Society.

[21] Shu-yu Guo and Jens Palsberg. 2011. The Essence of Compiling with Traces. In *POPL*. ACM.

[22] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *NDSS*. Internet Society.

[23] Renáta Hodován and Ákos Kiss. 2016. Fuzzing JavaScript engine APIs. In *Integrated Formal Methods*. Springer.

[24] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *USENIX Security*. USENIX.

[25] Gary Kwong. 2017. JavaScript Fuzzing in Mozilla, 2017. Slides online: https://nth10sd.github.io/js-fuzzing-in-mozilla/.

[26] Akash Lal and Shaz Qadeer. 2013. Reachability Modulo Theories. In *Reachability Problems*. Springer.

[27] K. Rustan M. Leino. 2008. This is Boogie 2. Online: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2.

[28] Sorin Lerner, Todd Millstein, and Craig Chambers. 2003. Automatically Proving the Correctness of Compiler Optimizations. In *PLDI*. ACM.

[29] Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *POPL*. ACM.

[30] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2018. Practical Verification of Peephole Optimizations with Alive. *Commun. ACM* 61, 2 (2018), 84–91.

[31] Mozilla. 2020. Security Severity Ratings/Client. Online: https://wiki.mozilla.org/Security_Severity_Ratings/Client.

[32] Mozilla Bugzilla 2021. Crash [@ ??] or Assertion failure: Expecting length to fit in int32, at jit/VMFunctions.cpp:2789. Online: https://bugzilla.mozilla.org/show_bug.cgi?id=1685925.

[33] Magnus O. Myreen. 2010. Verified Just-in-Time Compiler on x86. In *POPL*. ACM.

[34] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *SOSP*. ACM.

[35] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and Verification in the Field: Applying Formal Methods to BPF Just-in-Time Compilers in the Linux Kernel. In *OSDI*. USENIX.

[36] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-in-Time Compilation using Modular Control-Flow Integrity. In *CCS*. ACM.

[37] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-Preserving Mutation. In *S&P*. IEEE.

[38] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. NoJITsu: Locking down JavaScript Engines. In *NDSS*. Internet Society.

[39] Boris Shingarov. 2019. Formal Verification of JIT by Symbolic Execution. In *VMIL*. ACM.

[40] Window Snyder and Mike Shaver. 2007. Building and Breaking the Browser. Black Hat. Slides online: https://www.blackhat.com/presentations/bh-usa-07/Snyder_and_Shaver/Presentation/bh-usa-07-snyder_and_shaver.pdf.

[41] Maddie Stone and Jared Semrau an James Sadowski. 2024. We're All in this Together: A Year in Review of Zero-Days Exploited In-the-Wild in 2023. Online: https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/Year_in_Review_of_ZeroDays.pdf.

[42] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. 2020. Synthesizing JIT Compilers for In-Kernel DSLs. In *CAV*. Springer.

[43] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *ICSE*. IEEE.

[44] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *OSDI*. USENIX.

[45] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *OSDI*. USENIX.

[46] Chao Zhang, Mehrdad Niknami, Kevin Zhijie Chen, Chengyu Song, Zhaofeng Chen, and Dawn Song. 2015. JITScope: Protecting Web Users from Control-Flow Hijacking Attacks. In *INFOCOM*. IEEE.

[47] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *POPL*. ACM.