Cachet: A Domain-Specific Language for Trustworthy Just-In-Time Compilers

Michael Smith
UC San Diego

Abhishek Sharma
UC San Diego

John Renner
UC San Diego

David Thien
UC San Diego

Sorin Lerner
UC San Diego

Fraser Brown
CMU

Hovav Shacham
UT Austin

Deian Stefan
UC San Diego

Abstract

Just-in-time (JIT) compilers face a tough task: chewing up potentially-untrusted input and spitting out machine code that's at once fast, correct, and secure, working close-to-the-metal with low-level primitives in a tight time budget. Developers juggle complex invariants which generated code must respect, which can easily shatter the system's security model when broken. Cachet, our domain-specific language for JIT implementation, makes these invariants explicit and statically-verified. Our toolchain compiles Cachet code both to the SMT-solvable Boogie verification language and to C++ suitable for embedding in host applications and language runtimes. We are evaluating Cachet by reimplementing and verifying components of Firefox's JavaScript JIT.

Keywords secure compilation, domain-specific languages, just-in-time compilation

1 Introduction

The security of modern web browsers hinges on securely compiling and running enormous quantities of untrusted code, primarily JavaScript. Because JavaScript is such a dynamic language, browsers have adopted just-in-time (JIT) compilation techniques of increasing complexity in their JavaScript engines [2, 3, 11], to accommodate the performance demands of today's web applications. Browser JITs have been and continue to be an unending fount of exploitable bugs, to the point that Microsoft's Edge browser recently introduced a heightened security mode which disables JIT compilation entirely [10].

Securing these real-world just-in-time compilers against untrusted code presents a daunting challenge. They're sprawling black boxes that string arbitrary machine code together at the lowest level. Writing a secure production JIT requires not only writing fast, efficient, bug-free code, but writing code that *produces* code meeting all those properties as well. A clobbered register here, an insufficient type guard or bounds check there, and the security of the whole system comes tumbling down. Generated code must adhere to invariants that, currently, exist primarily in JIT developers' heads.

PriSC'23, January 21, 2023, Boston, MA, USA
2023.

Codifying these invariants and checking that they're upheld would be a first step toward securing the JIT. Some properties could be checked at runtime, by inspecting the generated code or intermediate compiler data structures. Cycles spent on runtime checks are cycles not spent running code, though, and since JIT compilation is ultimately a performance optimization, this limits the number and complexity of runtime checks that can feasibly be turned on at once. An ideal system would prove these invariants statically, offline, on the developers' machines and not the users'.

The production JIT engines we're interested in securing exist in a highly competitive ecosystem and have a fast-paced development model, so any proposed solution can only put so much drag on engineering velocity. Definitions of the intermediate bytecode languages in Firefox's JavaScript JIT have averaged a change every two days over the past two years [7–9]. Approaches requiring large auxiliary proofs would struggle to keep up and impose an untenable maintenance burden.

This also means that a solution must support partial, gradual specification and verification of the JIT system. A wholesale rewrite or proof effort would find itself woefully out of date by the time of its completion. To keep up with the moving target, one would need to start with a slice of the JIT and expand outward chunk by chunk, integrating intermediate results back into the production system and incrementally tightening invariants. Then maximizing embeddability and minimizing friction in interoperating with surrounding unverified code should be priorities.

Finally, we argue that a solution for implementing a maintainable verified JIT should look less like a proof system and more like a systems language: something familiar to the engineers already responsible for building and caring for the engine, and built on technologies they can hack on as needed. As much as possible, maintaining the JIT invariants should not be a separate process from maintaining the JIT implementation.

With this in mind, we present *Cachet*, a work-in-progress domain-specific language and toolchain which strives to meet these requirements. Cachet resembles a modern systems language (taking syntactic cues from Rust), but with first-class constructs for building secure compilers: expressing code transformations, control flow in dynamically-generated

```
ir CacheIR {
    // Bail out to the JavaScript interpreter if the given
        value is not null or undefined.
  op GuardIsNullOrUndefined(valueId: ValueId);
  /* ... */
}

ir MASM {
  // Test whether the JavaScript value is `null`.
  op BranchTestNull(cond: Condition, valueReg: ValueReg,
       label branch: MASM);
  // Test whether the JavaScript value is `undefined`.
  op BranchTestUndefined(cond: Condition,
       valueReg: ValueReg, label branch: MASM);
  /* ... */
}
```

Figure 1. Defining some SpiderMonkey IRs in Cachet.

```
compiler CacheIRCompiler for CacheIR emits MASM {
  op GuardIsNullOrUndefined(valueId: ValueId) {
    let valueReg = CacheIRCompiler::useValueId(valueId);

    // Create a bail-out path back out the JavaScript
        interpreter.
    CacheIRCompiler::addFailurePath(out label failure);

    label success: MASM;
    // Skip to `success` if the value is `null`.
    emit MASM::BranchTestNull(Condition::Equal, valueReg,
        success);
    // Bail out if the value is not `undefined`, either.
    emit MASM::BranchTestUndefined(Condition::NotEqual,
        valueReg, failure);
    // Mark the label `success` at the location following
        the branch instructions.
    bind success;
  }
  /* ... */
}
```

Figure 2. Defining a compiler from CacheIR to MASM.

```
interpreter MASMInterpreter interprets MASM {
  op BranchTestNull(cond: Condition, valueReg: ValueReg,
       label branch: MASM) {
    let value = MASM::getValue(valueReg);
    if cond == Condition::Equal {
      if Value::isNull(value) {
        goto branch;
      }
    } else if cond == Condition::NotEqual {
      if !Value::isNull(value) {
        goto branch;
      }
    } else {
      assert false;
    }
  }
  /* ... */
}
```

Figure 3. Defining an interpreter for MASM instructions.

code, and invariants in the form of statically-checked assertions. These assertions are checked not only against the compiler code, but also against the possible code the compiler can generate, given the definition of an interpreter for the target language (also written in Cachet). Cachet code can be extracted to efficient C++ by our toolchain, making it highly embeddable; external functions can be called directly, and gradually specified as needed or ported into Cachet themselves. We're currently evaluating Cachet by using it to write verified versions of components of SpiderMonkey, Firefox's JavaScript JIT: a compiler for a subset of CacheIR, SpiderMonkey's internal inline cache bytecode, as well as a verified range analysis for JavaScript (building on previous work [1]).

2 The Cachet DSL

```
struct NativeObject <: Object;

impl NativeObject {
  fn getFixedSlot(nativeObject: NativeObject, slot: UInt32)
       -> Value {
    let shape = Object::shapeOf(nativeObject);
    assert slot < Shape::numFixedSlots();
    unsafe { NativeObject::getFixedSlotUnchecked(
         nativeObject, slot) }
  }
  unsafe fn getFixedSlotUnchecked(
       nativeObject: NativeObject, slot: UInt32) -> Value;
  /* ... */
}
```

Figure 4. Wrapping an unsafe memory read with a statically-asserted bounds check.

We introduce a number of features of the Cachet DSL, using examples from our implementation of a verified compiler for CacheIR, an internal bytecode instruction set within SpiderMonkey. These CacheIR instructions are compiled to MacroAssembler (MASM), a SpiderMonkey-specific cross-platform assembly language which is specialized into platform-specific machine code further along the pipeline.

In Figure 1, we define these two IRs, `CacheIR` and `MASM`, and the signatures of their instructions. In Figure 2, we define a `compiler` which lowers `CacheIR` instructions to sequences of `MASM` instructions. The `MASM` instructions are appended to the instruction stream using Cachet's `emit` statement. Control-flow `labels` are declared and bound (with the `bind` statement) to points in the outgoing `MASM` instruction stream. These labels can be passed as inputs to `MASM` instructions like the branching tests in the example. Figure 3 shows the definition of a `MASM` interpreter which implements these branch instructions, using the Cachet `goto` statement to transfer control to the passed-in labels if the test results indicate the branches should be followed, and `assert`ing that the instruction is only used with supported `Condition` arguments. The `MASMInterpreter` translates from `MASM` instructions to higher-level operations on types in the JavaScript engine. In Figure 4, we specify one such operation, wrapping an unsafe memory read on `NativeObjects` performed by an opaque external function with a statically-checked assertion that the read is within-bounds for the object.

Given a chain of one or more compilation passes like in Figure 2 and an interpreter like in Figure 3, the Cachet toolchain verifies whether assertions like in Figure 4 can be violated for any inputs to any code possibly produced by the compiler, when interpreted by the given interpreter. This is accomplished by extracting a specially-structured Boogie [6] program from the input Cachet source code and posing it to the Corral [5] solver as a Reachability Modulo Theories (RMT) problem [4]. To make the solver converge despite the multiple levels of indirection at play, the Cachet toolchain **(a)** analyzes the potential control flow between emitted instructions and bound labels in the compiler, and **(b)** uses the results of this analysis pass to generate a version of the interpreter specialized to the control-flow structure of code generated by that compiler.

References

[1] 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 Conference on Programming Language Design and Implementation*. 135–150.

[2] Jan de Mooij. 2020. Warp: Improved JS performance in Firefox 83. https://hacks.mozilla.org/2020/11/warp-improved-js-performance-in-firefox-83/

[3] Jakob Gruber, Leszek Swirski, and Toon Verwaest. 2022. Maglev. https://docs.google.com/document/d/13CwgSL4yawxuYg3iNlM-4ZPCB8RgJya6b8H_E2F-Aek/edit

[4] Akash Lal and Shaz Qadeer. 2013. Reachability Modulo Theories. In *7th International workshop on Reachability Problems (Invited Paper)* (7th international workshop on reachability problems (invited paper) ed.). https://www.microsoft.com/en-us/research/publication/reachability-modulo-theories/

[5] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. 2012. Corral: A Solver for Reachability Modulo Theories. In *Computer-Aided Verification (CAV)* (computer-aided verification (cav) ed.). https://www.microsoft.com/en-us/research/publication/corral-a-solver-for-reachability-modulo-theories-2/

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

[7] Mozilla. 2022. CacheIROps.yaml file revisions. https://hg.mozilla.org/mozilla-central/log/tip/js/src/jit/CacheIROps.yaml

[8] Mozilla. 2022. LIROps.yaml file revisions. https://hg.mozilla.org/mozilla-central/log/tip/js/src/jit/LIROps.yaml

[9] Mozilla. 2022. MIROps.yaml file revisions. https://hg.mozilla.org/mozilla-central/log/tip/js/src/jit/MIROps.yaml

[10] Johnathan Norman. 2021. https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode/

[11] Assaf Sion. 2021. The mysterious realm of JavaScriptCore. https://www.cyberark.com/resources/threat-research-blog/the-mysterious-realm-of-javascriptcore