WebAssembly Instrumentation Framework Design 2018-01-25

Design Decisions

Streaming Instrumentation?

WASM allows streaming parsing and compilation, i.e., start while only parts of the module have been transmitted over the network (useful for browsers, especially on mobile).

Pros:

- not possible for x86 but for WASM
- can execute parts even before instrumentation is complete, i.e., lower instrumentation latency
- new "term"

Cons:

- implementation: either in-engine or as a proxy
- ??? only useful if instrumentation is faster then network speed, otherwise the latter catches up anyway
- actual use in practice? ??? a developer will have the modules local anyway, not many will be doing instrumentation in production, especially not on mobile

Parallel Instrumentation? (related to Streaming)

Sections in WASM modules can be decoded/compiled independently of each other, so parallelize instrumentation per section/function.

Pros:

- not possible for x86 but for WASM
- higher instrumentation bandwidth
- new "term"
- unlike streaming instrumentation, can be implemented externally / as a standalone "WASM module transformer"

Cons:

- ??? multithreading is hard to implement (?)

MP: How about single-pass, modular instrumentation? I.e., it could be used for streaming instrumentationand it could be easily parallelized.

Record/Replay?

As per its paper, Jalangi does this, i.e., it first records a trace of all memory accesses/used values and later uses them in the replay phase for shadow execution (i.e., the actual analysis on top of the instrumentation).

Pros:

- useful for time-traveling debugging?
- potentially the recording is quicker than very slow analyses, so the instrumented binary would run quicker than with "online" analysis

Cons:

- ??? Jalangi2 doesn't use this anymore (why?)
- at least doubles the memory requirements of the instrumented binary cf. the uninstrumented one

- is the analysis really more expensive than recording every memory access? (I suspect not, since the analysis results are probably much less data to store/process.) So in reality, does this slow down the instrumented binary more than "online" analysis?

MP: Maybe as a recond step, once we have an instrumentation tool?

Static vs. Dynamic (JIT) Instrumentation

Dynamic: instrument only "on-demand" when some code is executed (== jumped to), not everything beforehand. PIN and Valgrind do it.

Pros Dynamic:

- necessary for x86, since static disassembly is hard / impossible
  (linear sweep is not reliable because a) code and data are mixed and b) variable-length instructions / very dense ISA)
- only instruments the parts that actually get executed
- change instrumentation at runtime (e.g., add more specific instrumentation at runtime or disable after some time)

Pros Static / Cons Dynamic:

- robust disassembly in WASM is possible (it is designed exactly for that)
- JIT machinery is complicated:
  - needs instrumented code cache HashMap + some quick lookup before that
  - need to include instrumentation framework at runtime of the instrumented binary
  - needs to patch code at runtime, i.e., change jump targets from the "dispatcher" to the newly instrumented code (this is not even possible in WASM!?)
- ??? most of a binary is probably executed, thus dynamic instrumentation only delays the instrumentation but has to do it eventually anyway (TODO: evaluate! is this true?)

MP: Sounds like static makes more sense for WASM.

Implementation of the Instrumentation: In-Browser/In-Engine vs. External Program

Pros In-Browser:

- can reuse existing parts of the codebase, e.g., WASM decoder (?)
- necessary for streaming instrumentation

Pros External:

- easier to get going: no need to modify the engine, recompile browser everytime, read and find around their source code
- we cannot compete with Browser vendors anyway, so rather not try
- users do not need to install custom browser to use instrumentation
- we do not need to track changes in the browser
- works across different browser and different vendors

MP: If implemented as an external tool, couldn't we compile it to WASM and then run in the browser anyway? I'd go for external.

Disassemble & Resynthesize vs. Copy & Annotate

I think Valgrind paper has coined these terms: D&R means having your own IR to which the binary is parsed first, then the instrumentation added in this IR, and then serialized to a binary again. In particular, the original binary is discarded completely and everything only works on the IR. C&A implies that the original binary's code is copied over and instrumentation inserted in between. The distinction doesn't seem 100% clear to me, especially since PIN (which the Valgrind paper says is C&A) also rewrites original binary instructions (they do register reallocation).

The distinction in WASM is even less clear: WASM is executed on a VM/JIT anyway.

Pros D&R:

- original code and instrumentation code are optimized together, e.g., calls in both cases could be inlined by the JIT during execution
- for x86: less register spilling, i.e., the analysis code can use registers just as good as the original code, whereas in C&A the analysis code must make sure not to modify any registers of the original binary -> lots of spills
- for WASM: there is already a well-specified, small IR (unlike for x86, where this is a major challenge to correctly capture all of the ISA's behavior)
- for WASM: parsing binary to an IR and serializing again should be easy to make round trip (unlike for x86)

Pros C&A:

- for x86: more robust, less danger of changing semantics of an instruction if the original instructions are executed unchanged
- for x86: less implementation effort, no need to capture the huge CISC instruction set.

MP: We should use an IR, either an existing one or our own (the latter might also become a contribution).

"Heavy" vs "Light" Instrumentation

Valgrind paper argues their framework enables analyses that are not possible with "lightweight" PIN/DynamoRIO, e.g., Memcheck (TODO: what does it actually do?). Where is the boundary between lightweight and heavyweight?

Pros Heavy:

- more complex analyses possible

Cons Heavy:

- performance
- larger API to implement (?)

MP: We should aim for a general framework that supports many different, possibly complex analyses. Performance definitely also matters, but should be a secondary goal.

API Design: based on Event Streams (and Filters) vs. Callbacks vs. ???

RoadRunner paper talks a lot about their API design: Analyses are implemented as "filters", i.e., methods that capture some events (generated by the framework, e.g., `FieldAccessEvent`) and give the uncaptured events through to later "filters" in the "tool chain".

MP: We could also have a low-level API, which provides access to basic events, and a high-level API, which summarizes low-level events into higher-level events. To make more concrete suggestions, I'll have to learn more about WASM.

Language Choice: Instrumentation Framework

C++:

- already exists some tooling (WABT)
- fast (?)
- I am not an expert in C++ :(

Rust:

- lots of small community for WASM already: a parser/unparser, a small interpreter, maybe Mozillas internal implementatioN (?)
- easier/safer for multithreading stuff than C++
- I already know the language a bit (and I love it) :)

JavaScript:

- huge community, close to the web world
- very high level
- possibly slow
- no chance of every intergrating that inside the engines of the browsers :()

Language Choice: User Analyses

WASM:

- easy to insert in the existing binary then :)
- too low-level to implement large analyses in it

JavaScript:

- how expensive is the JS-WASM interop? If it is expensive, instrumentation could become too slow
- JavaScript itself is already slow

C++, Rust, any language that compiles to WASM:

- fast, but still high-level

MP: Can't we compile JS to WASM? Any language that compiles to WASM seems fine.

Other

- WASM has only structured control flow: is this something new/noteworthy?

Possible Analyses / Use-Cases for Instrumentation

Simple ones, i.e., analysis itself is well known

- instruction counting: which instruction is executed how often
  1. per whole module (simplest case)
  2. per function
- basic block counting / coverage
  - needs some control-flow analysis
- taint analysis
  - simplest version: just 1 bit, "tainted" or not
  - needs "shadow values", i.e., associated meta information for each memory "cell" (byte?)
  - needs "shadow execution", i.e., computation on this meta information, invoked when computation on actual values happens
- memory access tracing
  - record address of each read/write
  - show regions that are never accessed (maybe as an image?) `->` should have not been allocated?
  - show regions that are accessed very often `->` potential for optimization?
- call counting
  1. which function is called how often
     - need to handle direct and indirect calls
  2. from which other function (build flame graph?)
- null/empty instrumentation: run instrumentation framework but do no user analysis

Known but not as simple to implement:

- general time profiling: which function is executed how long, flamegraph?
  - precise (actually measure time) vs. sampling (periodically observe which is currently executing, is this possible in WASM?)
- fault injection: flip bits/change bytes in WASM memory
- concolic execution (?)
- track origin of null values: when null value is read, where was it written from?

New analyses, possibly *specific to WebAssembly*:

- allocation profiler: invocations of `Memory.grow(num_pages)`
  - which function (as a flamegraph)
  - how often
  - with what page count
- JavaScript-WASM interop linter
  - correctness: do JavaScript types and WebAssembly types (mis)match? (do other instrumentation first, maybe I learn about interop in the process anyway)
  - performance: how long does the conversion take?
- data type mismatches in WebAssembly: detect pointer arguments and remember how often how many bytes are read from this ptr, if different sizes `->` possible data type mismatch?

Other:

- record & replay, for time-traveling debugging