WASM

- Goal: run fast code in web applications
- Javascript is universally supported but not a great fit in some cases
  - Need high performance and want global support for many different languages
- Challenge: code must be isolated
  - Code from one website cannot tamper with data from another site / on the computer

---

- In the last lecture we saw many ways that the OS can be used to add isolation
  - They require special privileges and some only run on certain browsers
  - VMs don't work well on CPUs without direct hardware VM support

Software Fault Isolation

- Limits the effects of buggy code just to the application itself
  - Can limit effect to just a user-defined boundary (i.e. module / library)
- Does not rely on hardware / OS support
- Requires cooperation between developers and devices
  - Cannot take existing binary and run it in isolation
  - On the other hand, containers / VMs can run existing Linux applications

WASM Description

- Can take functions written in any language and compile it into wasm
- Many browsers can run wasm with some wasm runtime to natively execute it
- Looks very much like machine code
- General approach: sandbox which memory the wasm code can access
  - This occurs when the wasm runtime is converting wasm code into native instructions
  - add / sub are directly translated to native instructions
  - load / store just needs to check if address is within bounds
    - Just adds an if statement before the load / store
  - jumps are more challenging because we could jump past instrumentation / these if statements so we need to make sure we only jump to instrumented code
    - To get around this, we define a table of all possible jump locations and then all indirect jumps just supply the index to jump to
- These checks for the indirect jumps can also be done at compile time, but having these if statements hurts performance
- For global / local variables just figure out how many there are and then we can keep an array of them and just make loading them as simple as load with an offset
  - The compiler in the wasm runtime has to verify these are all correct before it translates into x86 and runs it
- For the stack, we can figure out how much its stack will potentially grow to and then also just use indexing to grab the proper stack module
- Heap accesses still have to just check against the total size allocated to the sandbox
  - This is the biggest overhead to wasm code
  - Some optimizations:
    - Batch these checks if you have multiple load statements
    - We could also use page tables to optimize this by mapping the possible region into virtual memory