DERIVCO

Derivsc[h]ool

ESTONIA

What is WebAssembly?

Daniel Priori

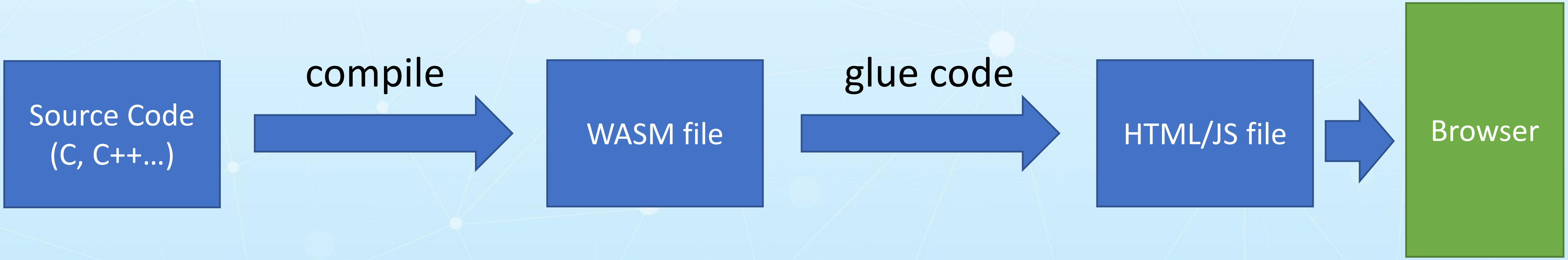
Topics for today

- What is WA exactly?
- Why?
- WASM architecture
- Advantages
- How it works
- asm.js VS wasm
- Some features
- Process to compile
- Roadmap
- Use cases
- Examples, links and demos
- Let´s try!

WA
WEBASSEMBLY

What is WA exactly?

- Source code compiled and accessed from browser that can achieve a performance close to a native application
- C, C++, Rust, Go (for now) source code
- Compiler Target (code generated by compilers)
- Binary format
- Use inside web browsers with Javascript
  - Load the binary code using javascript and can access and manipulate
- Performance close to native code (today 1.2x)



What is WA exactly?

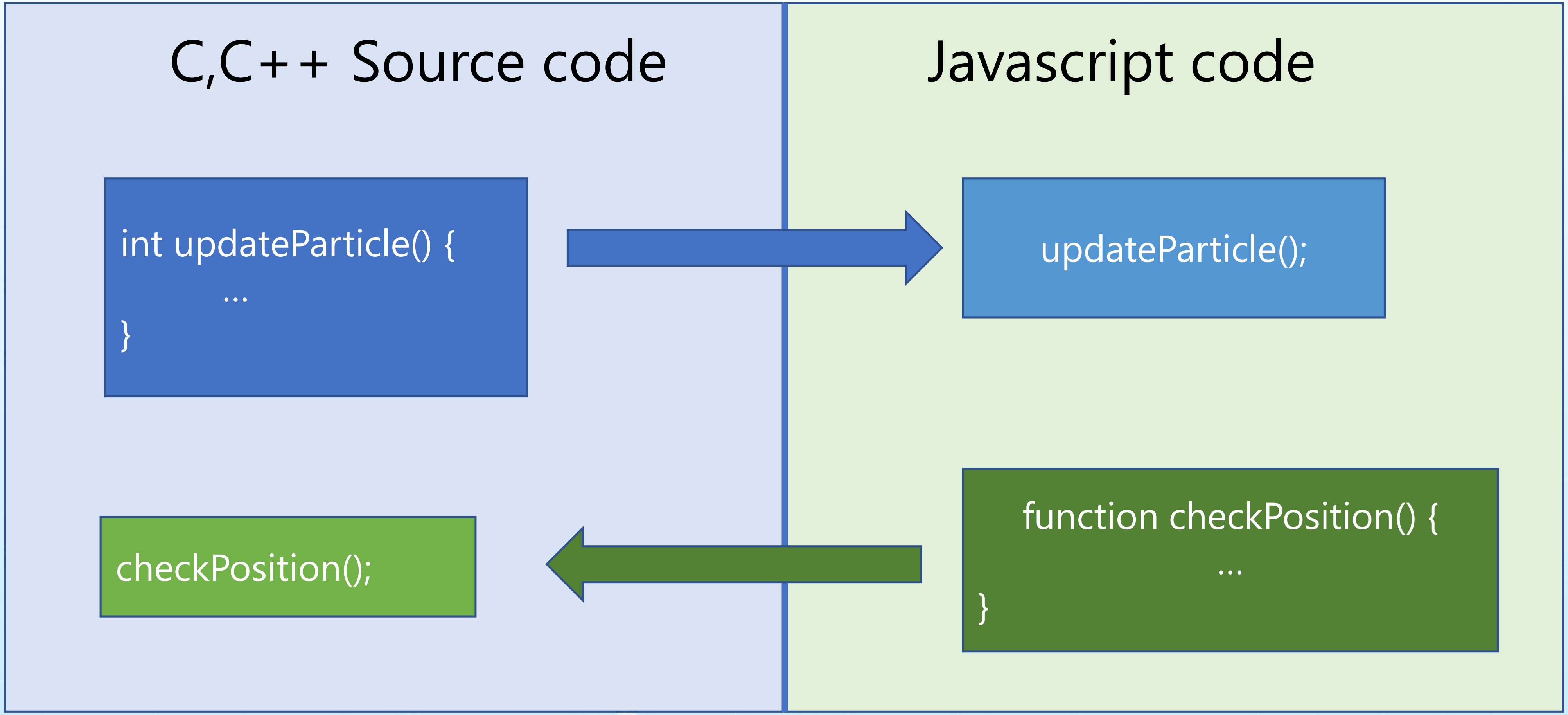
It will replace javascript, right?

- nooop!
- It will NOT substitute JavaScript → it will to work alongside it.

Performance of typed/compiled languages + JavaScript´s Flexibility

- With WebAssembly API on JavaScript, you can load modules WASM on JS application and share functionalities between them!

What is WA exactly?



Why?

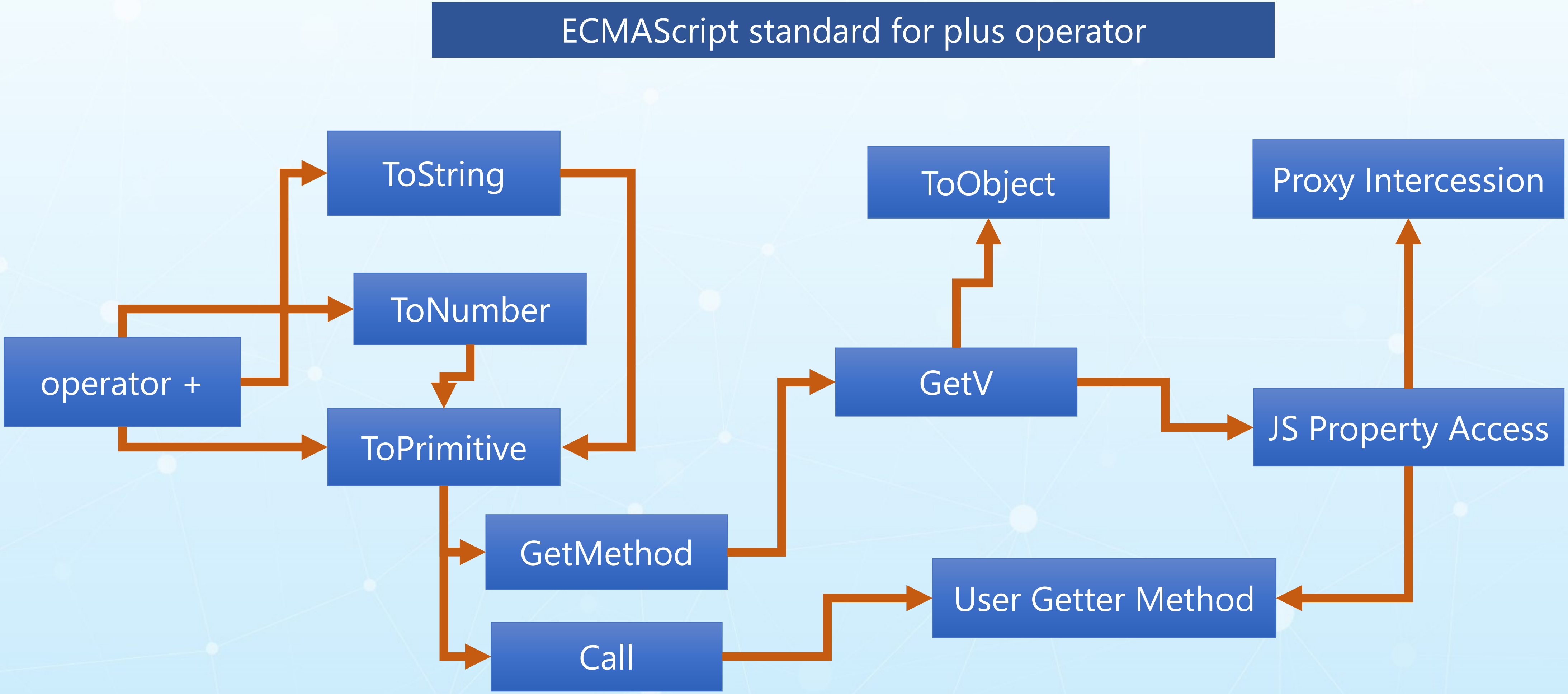
- Performance
  - Today: a lot of effort to achieve tiny optimization
  - JS dynamically typed language
    - Limit to push optimization
    - JS is a bottleneck
- Native source code
  - Able to cross-compile the same source code and deploy in a browser

DERIVCO

Why?

```
function add(a, b) {
    return a + b;
}
```

Why?


ECMAScript standard for plus operator
ToString
ToObject
Proxy Intercession
ToNumber
operator +
GetV
JS Property Access
ToPrimitive
GetMethod
User Getter Method
Call

Why?

- The aim:
  - Write a + b → convert into a single CPU instruction

WASM architecture

- A stack machine, 4 types and void, 67 instructions
- Designed to support streaming compilation
- Simple validation rules
- Exports/ imports functions
- Linear memory is shared with Javascript

WASM architecture

- **Data Types**
  - void i31 i64 f32 f64
- **Functions**
  - Flat, single global table
  - Static binding
  - Indirect calls through table
- **State: linear memory**
  - Large, bounds-checked array
- **Trusted execution stack**

- **Data Operations**
  - i32: + - * / % << >> >>> etc
  - i64: + - * / % << >> >>> etc
  - f32: + - * / % sqrt ceil floor
  - f64: + - * / % sqrt ceil floor
  - conversions
  - load store
  - call_direct call_indirect
- **Structured Control Flow**

Advantages - general

- **Languages**: not just javascript but you can use many languages
  - C, C++, Rust and Go for now. But since the team will add more features, more languages will be available
- **Reused code**: some applications were created using those languages and instead of rewriting to Javascript, we can use the quite same and just use it inside browsers.
- **Performance:** close to native code
- **Security**: is the same as Javascript since it use the same linear shared memory

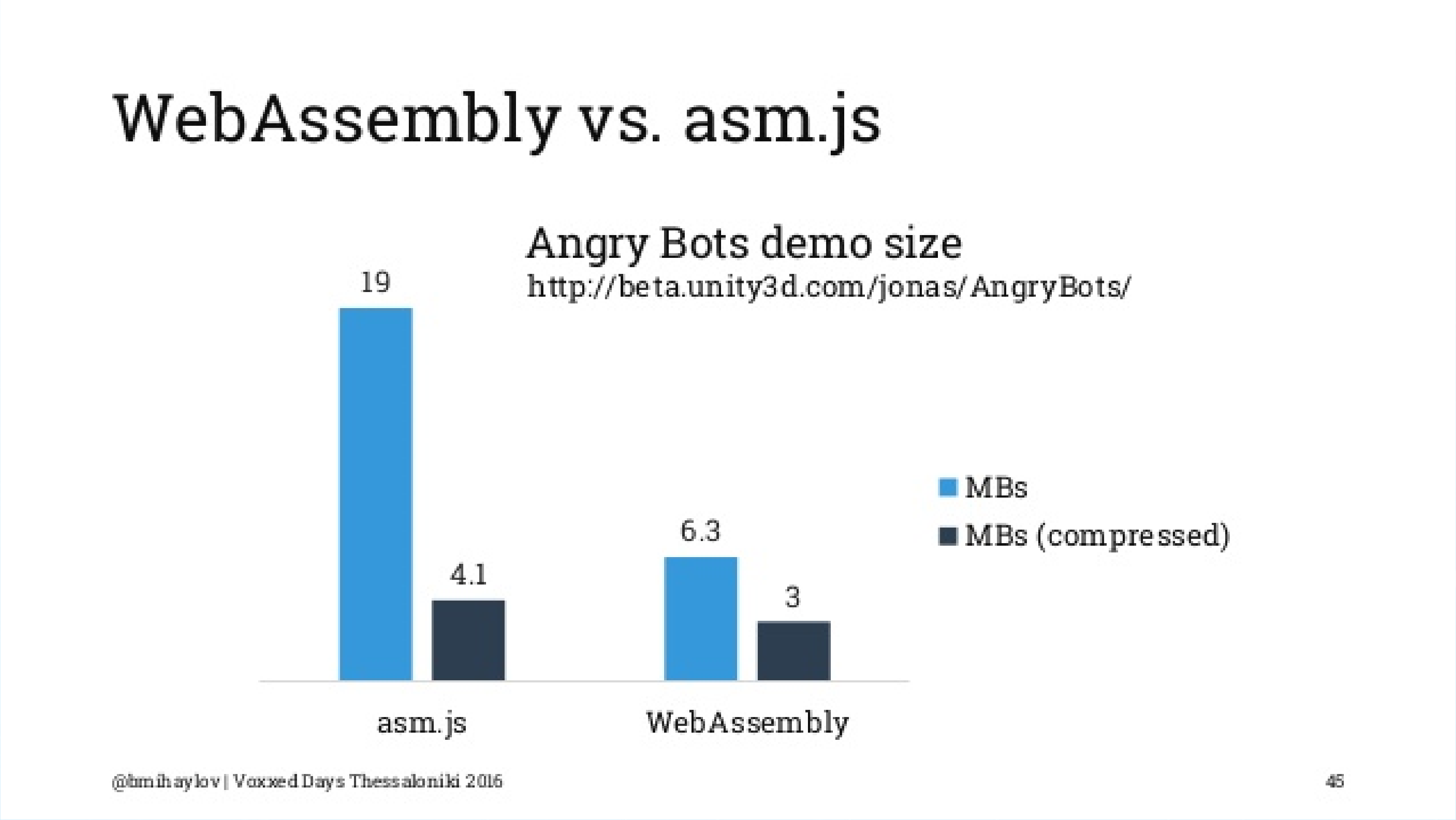
Advantages - technical

- **Loading Time**: Javascript slower than WASM
  - wasm → binary format (call instructions directly).
  - wasm file can be **smaller** than asm.js files (can reach 50%)
- **Parser**: there is no need for parsing .wasm codes.
  - Decode WASM file → less time (is already closer to machine code)
- **Compiling/ Optimization**: is optimized when binary code is generated → no need to spend time when executing.
  - Javascript uses JIT to optimize and execute.

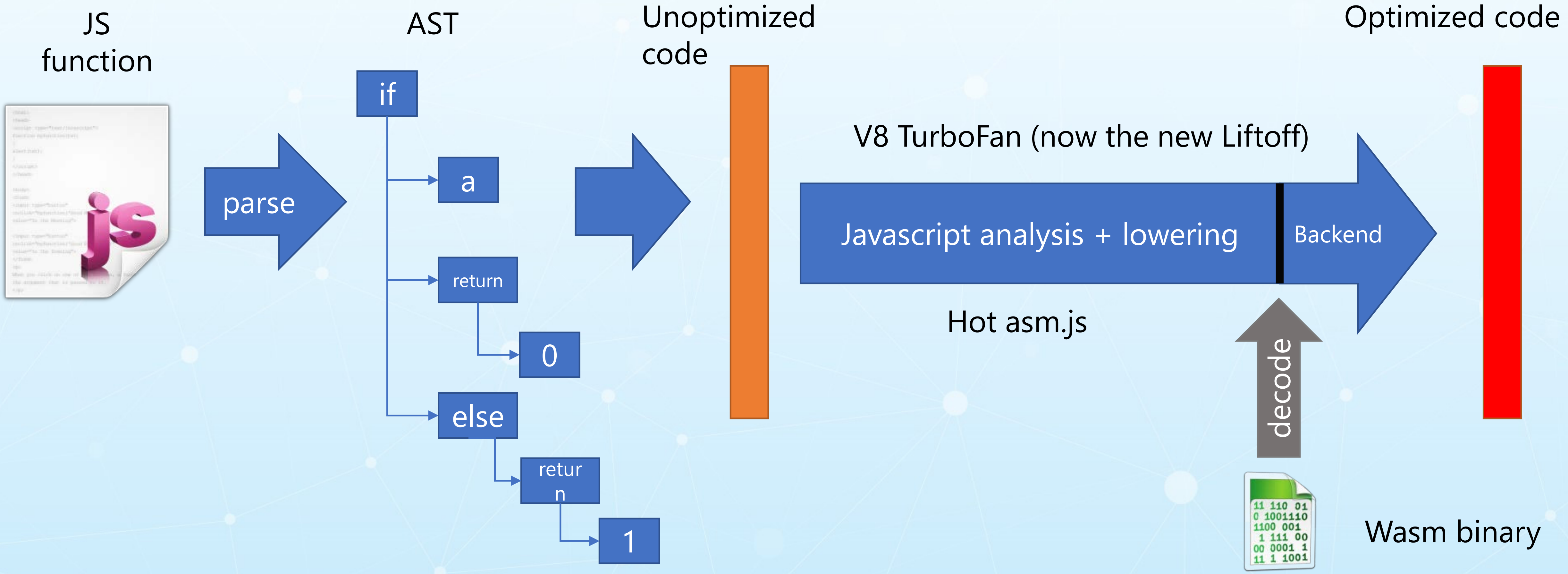
Advantages - technical

- **Execution**: is part of execution and optimizations.
  - Wasm doesn't need these optimizations since it's generated from statically typed code.
- **GC**: not yet but it's in the roadmap.
- **Binary encoding of the AST (Abstract Syntax Tree):** that the parser calculates. It has 2 big benefits:
  - The JS engine can skip the parsing step
  - It's much more compact than the JS original source

Advantages - technical



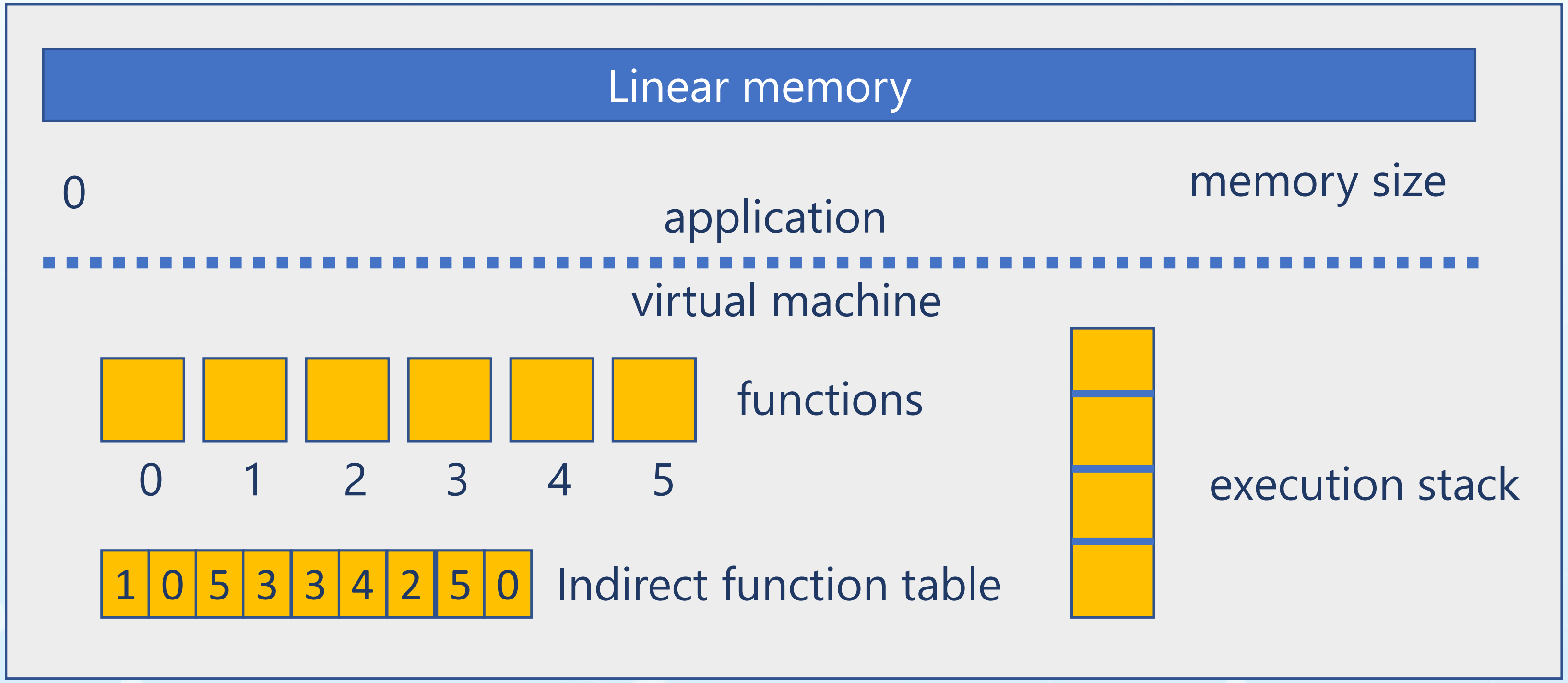
How it works?



JS function
parse
AST
if
a
return
0
else
retur n
1
Unoptimized code
V8 TurboFan (now the new Liftoff)
Javascript analysis + lowering
Backend
Hot asm.js
decode
Optimized code
Wasm binary

How it works? - Security

- Execution stack separate from the linear memory
- There is no way you can modify inside it and change things like variables.
- Functions use integer offsets rather than pointers
- Functions point into an indirection function table
- Those direct, calculated numbers jump in the function inside the module



DERIVCO

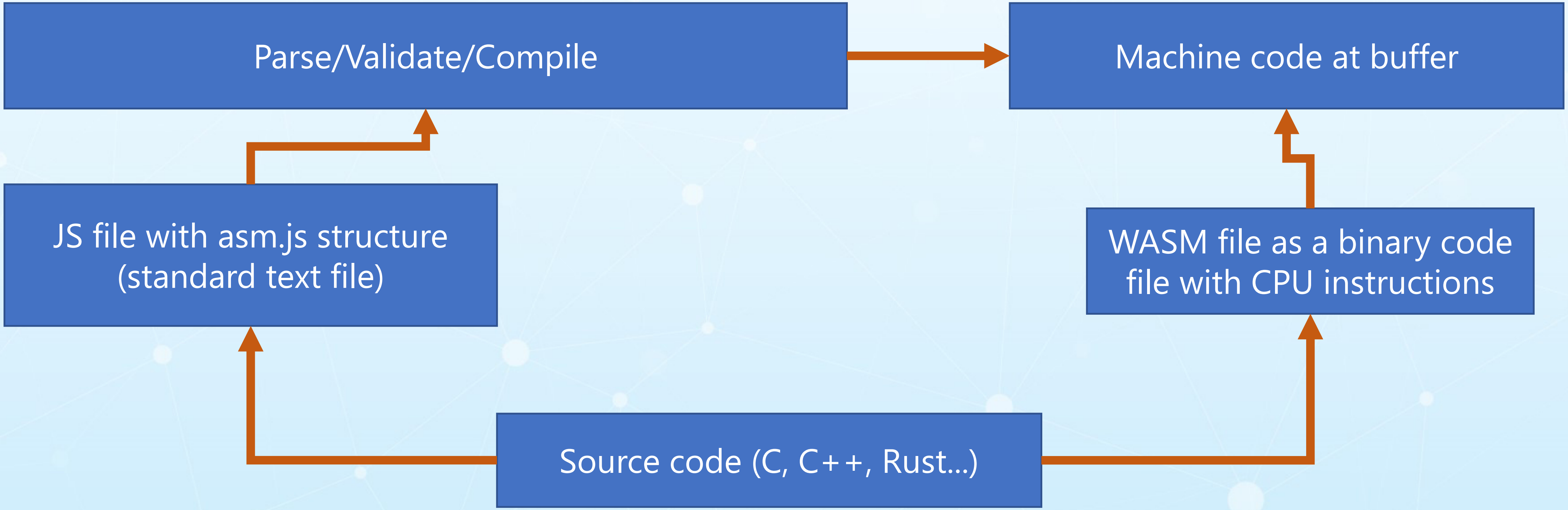
asm.js VS wasm

- **asm.js** it takes the source code (C++) compiles it down into a tiny subset of JS with "highly optimizable" instructions (annotations) and JS engines attempt to compile into a fast native code.
  - Declare the type (int, float) and the js engine will execute the instructions faster.
  - It uses Ahead-of-Time validation and compilation for asm.js
  - It generates a JS file (asm.js) as a text format.
- **wasm** is a binary format from the source code. It's not a bytecode.
  - smaller file size (it calls CPU instructions directly)

asm.js VS wasm

```
function add(a, b) {
    a = a | 0; // a : int
    b = b | 0; // b : int
    return (a + b)|0;
}
```

asm.js VS wasm



DERIVCO

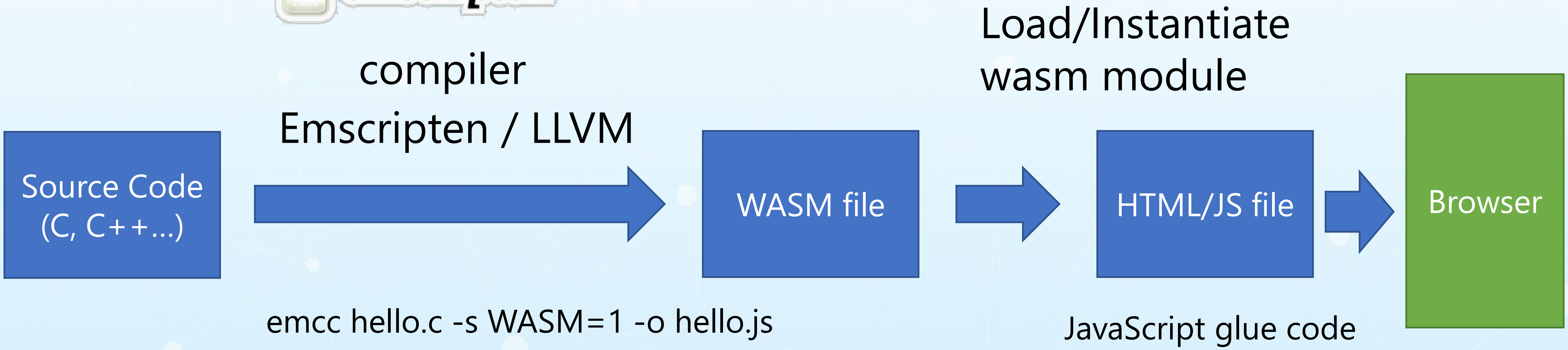
Some features

- **WebAssembly.Global**
  - global variable instance – accessible from all wasm module instances and JS
- **WebAssembly.Memory**
  - Memory object which is a resizable ArrayBuffer that holds the raw bytes of memory
- **WebAssembly.Table**
  - object that stores function references
- **WebAssembly.CompileError** – at compile time
- **WebAssembly.LinkError** – at instantiation time
- **WebAssembly.RuntimeError** – at runtime

DERIVCO

Process to compile

emscripten




```
emcc hello.c -s WASM=1 -o hello.js

em++ hello.cpp -s WASM=1 -o hello.js --bind --std=c++11
```

Roadmap

WASM Goals

Open standard inside the W3C WebAssembly Community Group with the following goals:

- Be fast, efficient, and portable
- Be readable and debuggable
- Keep secure
- Don't break the web

Roadmap

- Threads
- SIMD
- GC

- https://webassembly.org/docs/future-features/

Use Cases

- Better execution for languages and toolkits that are currently cross-compiled to the Web (C/C++, GWT, …).
- Image / video editing.
- Games:
  - Games that need to start quickly. ;)
  - AAA games that have heavy assets/calculations/management.
  - Game portals (mixed-party/origin content).
- Peer-to-peer applications (games, collaborative editing, decentralized and centralized).
- Music applications (streaming, caching).
- Image recognition.
- Machine Learning / Artificial Intelligence apps
- Live video augmentation (e.g. putting hats on people's heads).
- VR and augmented reality (very low latency).
- CAD applications.

Use Cases

- Scientific visualization and simulation.
- Interactive educational software, and news articles.
- Platform simulation / emulation (ARC, DOSBox, QEMU, MAME, ...).
- Language interpreters and virtual machines.
- POSIX user-space environment, allowing porting of existing POSIX applications.
- Developer tooling (editors, compilers, debuggers, ...).
- Remote desktop.
- VPN.
- Encryption.
- Local web server.
- Common NPAPI users, within the web's security model and APIs.
- Fat client for enterprise applications (e.g. databases).

DERIVCO

Examples, links and demos

WASM Tools

https://mbebenita.github.io/WasmExplorer/

https://webassembly.studio/

https://cdn.rawgit.com/WebAssembly/wabt/e0719fe0/demo/

https://www.npmjs.com/package/cpp-wasm-loader

WASM Tutorials, More Details

https://developer.mozilla.org/en-US/docs/WebAssembly/

https://webassembly.org

https://codelabs.developers.google.com/codelabs/web-assembly-intro/index.html?index=..%2F..%2Findex#0

http://kripken.github.io/emscripten-site/docs/getting_started/Tutorial.html

Examples, links and demos

WASM Demos, articles and use cases

https://maierfelix.github.io/wasm-particles/static/

https://hackernoon.com/games-build-on-webassembly-3679b3962a19

https://d2jta7o2zej4pf.cloudfront.net/

https://s3.amazonaws.com/mozilla-games/ZenGarden/EpicZenGarden.html

https://s3.amazonaws.com/mozilla-games/tmp/2017-02-21-SunTemple/SunTemple.html

https://s3.amazonaws.com/mozilla-games/tmp/2017-02-21-StylizedRendering/StylizedRendering.html

https://s3.amazonaws.com/mozilla-games/tmp/2017-02-21-PlatformerGame/PlatformerGame.html

https://github.com/kripken/emscripten/wiki/Porting-Examples-and-Demos

https://pspdfkit.com/blog/2018/a-real-world-webassembly-benchmark/

Let´s try

1. Install tools to compile
   1. Emscripten and dependences
      1. https://kripken.github.io/emscripten-site/docs/getting_started/downloads.html
2. Create a simple C/C++ code
3. Compile using emscripten and/or web tools
4. Create HTML/JS glue code