How Is The Sausage Made?

A Whirlwind Tour of V8, Real-World JIT-Compilers, and Their Trade-Offs

Daniel Lehmann
V8 Team

Program Analysis Course, University of Stuttgart

January 2024

Abstract

V8 (https://v8.dev) is a high-performance JavaScript and WebAssembly engine, used in Chrome and Node.js. It has multiple tiers, to squeeze out most in the complex trade-off space between execution speed, compilation time, generated code size, security and other factors. This talk will cover fundamentals of JIT (just-in-time) compilers and give a brief overview of the full range of programming language implementation strategies present in V8: an interpreter for low latency, non-optimizing but fast baseline compilers, an optimizing compiler with a single IR (intermediate representation), and a highly optimizing multi-pass compiler as the "top-tier". We will see how V8 has evolved over time and how each of those tiers is motivated by improvements on benchmarks and real-world websites. We will also briefly discuss practical aspects of developing and maintaining a 1M+ lines code base, running on 4+ operating systems and 7+ hardware architectures, and how to deliver not just fast but also robust and secure software to billions of users.

<meta>

- 🔎 My experience / background: V8, WebAssembly, performance optimizations
  - But the basic principles are the same in other engines / virtual machines!
- 🤔 Please interrupt at any time for questions!
- 🤸 Interactive: I will ask some (simple) questions myself :-)
- 💬 Please reach out afterwards, always happy to get feedback!

What is V8?

https://v8.dev

- High-performance JavaScript and WebAssembly engine
- Used in Google Chrome, Node.js, and Chromium-based projects
- Open-source: v8.dev/docs/source-code
  - ~1.2M lines of C++ (+ tests)
  - Many people contributing (Google, other companies, individuals)

- Lots of interesting engineering challenges!
  - 6+ different interpreters/compilers
  - Runs on 4+ operating systems, 7+ hardware architectures

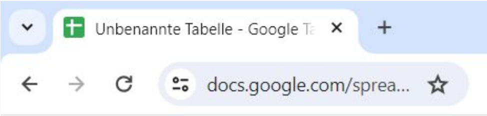
📋 Take-Home Points

I. **Background, terminology**
   - What is a *just-in-time compiler*? Why *multiple tiers*?
   - What does it mean to be *high-performance*?
   - Techniques: *speculative optimizations*, *deoptimization*, *unboxing*, …

II. **Concrete example:** V8
   - Overview of the *execution / compilation pipeline*
   - Different tiers, e.g., *Ignition*, *TurboFan*, *Liftoff*, *Maglev*

III. **Practical considerations**
   - Software engineering, performance, security, …

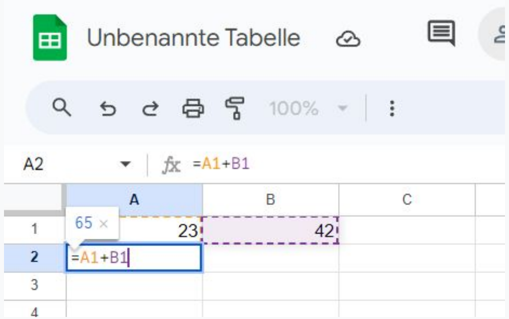
How is JavaScript executed?



Loads /
contains

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

Executes



V8 Overview

V8 has multiple **tiers**:

JS

- Interpreter *Ignition*
- Baseline compiler *Sparkplug*
- Mid-tier compiler *Maglev*
- Top-tier compiler *TurboFan*

WA

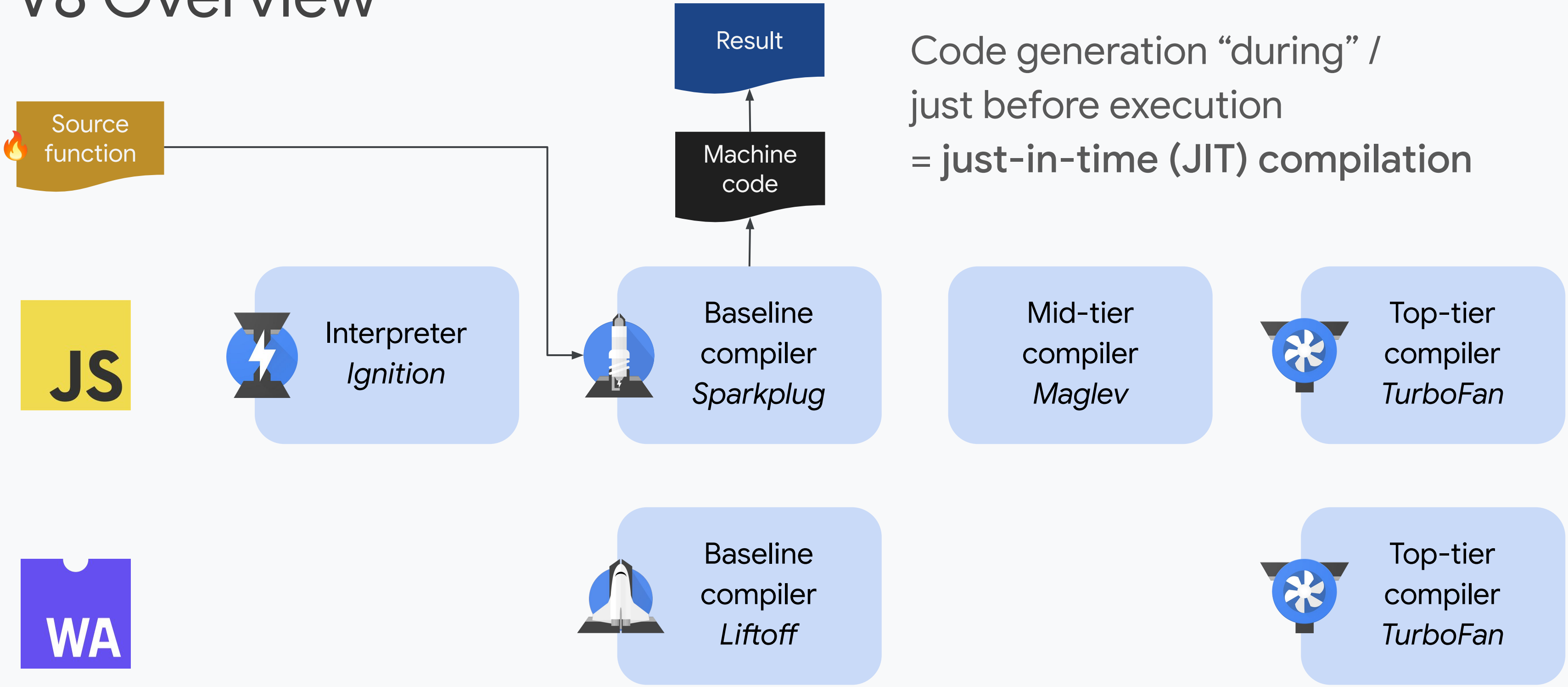
- Baseline compiler *Liftoff*
- Top-tier compiler *TurboFan*

V8 Overview

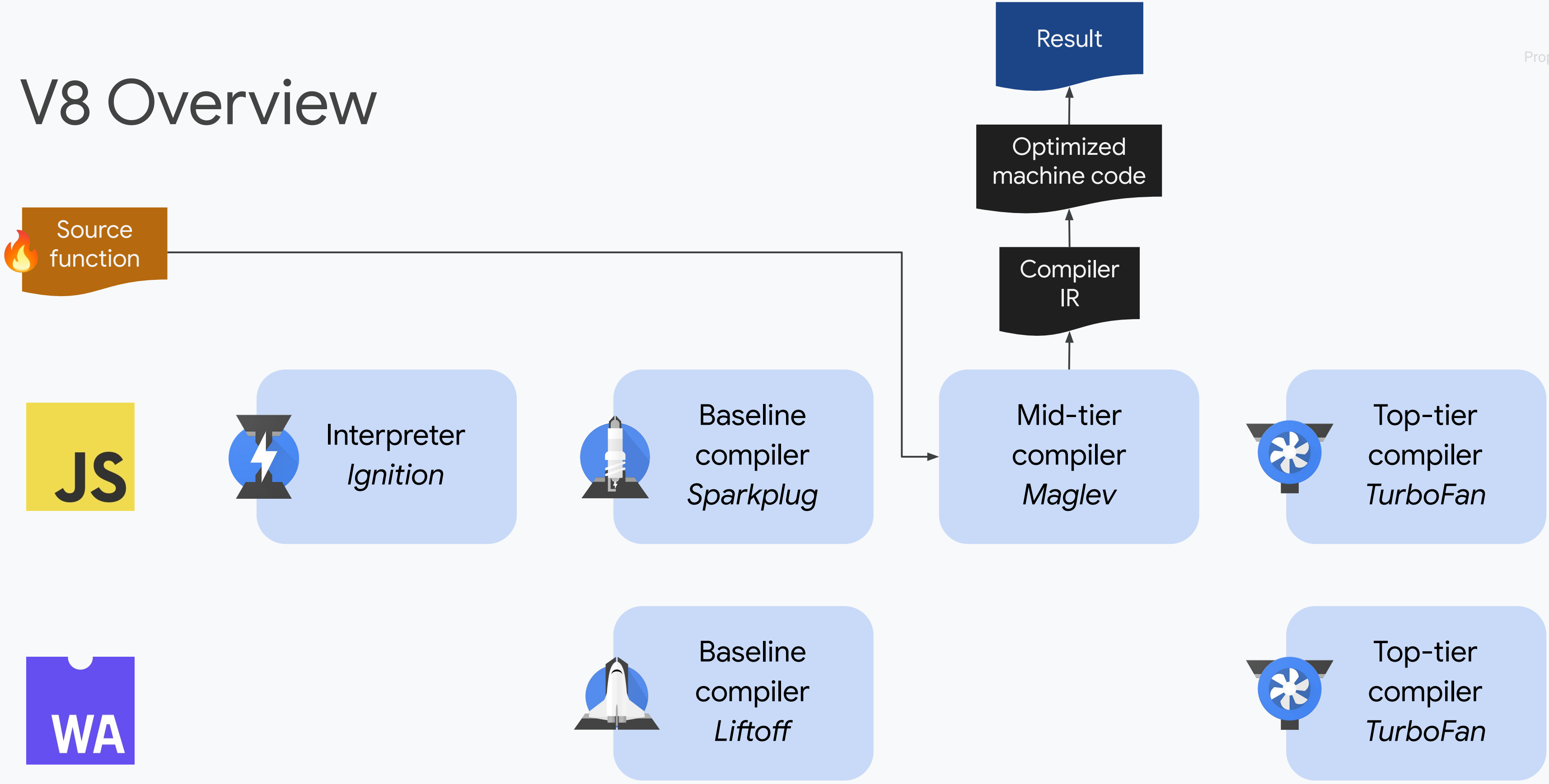

Source function
Result
JS
Interpreter *Ignition*
Baseline compiler *Sparkplug*
Mid-tier compiler *Maglev*
Top-tier compiler *TurboFan*
WA
Baseline compiler *Liftoff*
Top-tier compiler *TurboFan*

V8 Overview

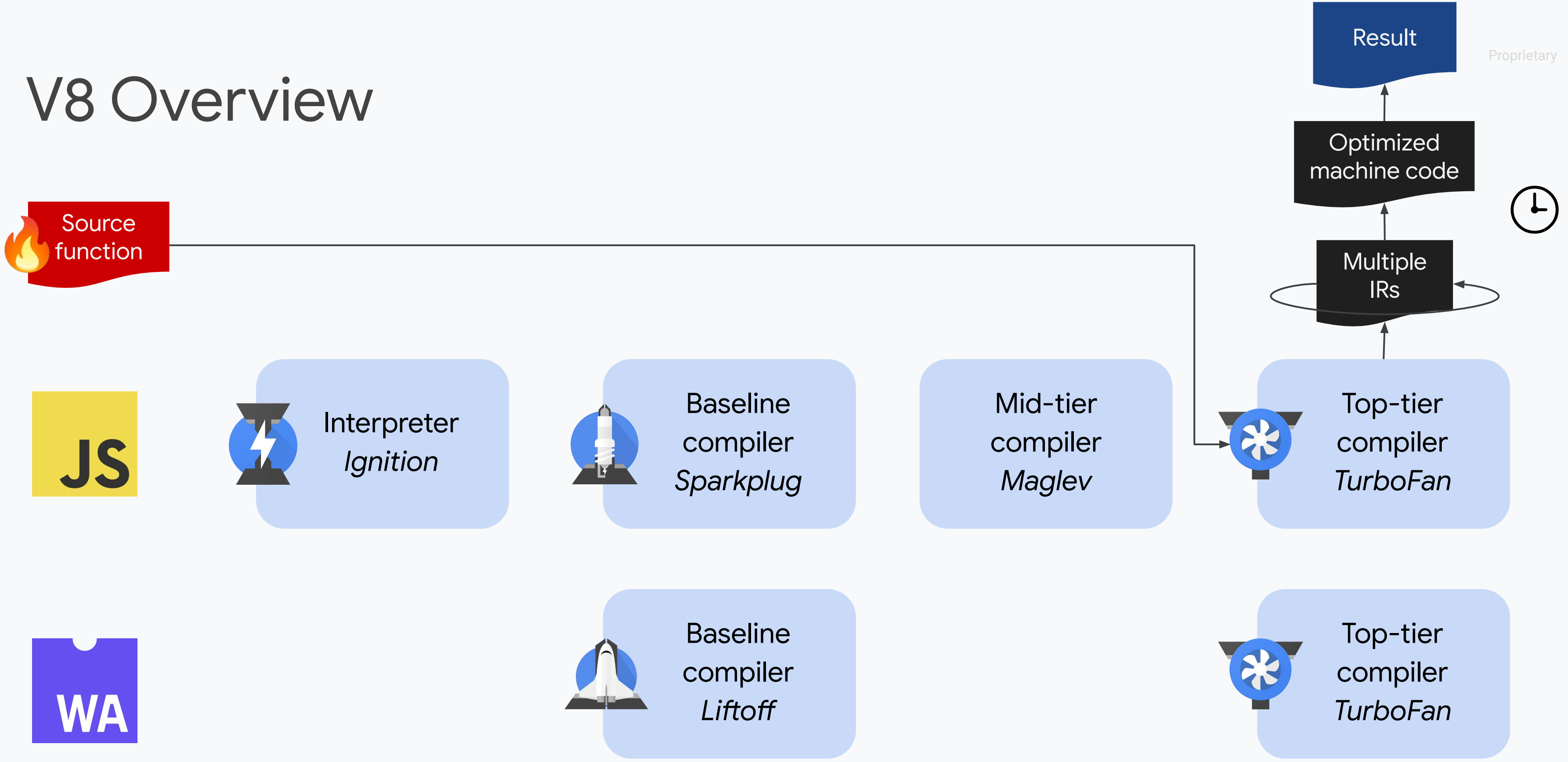
Code generation “during” /
just before execution
= **just-in-time (JIT) compilation**



V8 Overview


Result
Optimized machine code
Compiler IR
Source function
JS
Interpreter Ignition
Baseline compiler Sparkplug
Mid-tier compiler Maglev
Top-tier compiler TurboFan
WA
Baseline compiler Liftoff
Top-tier compiler TurboFan

V8 Overview


Result
Optimized machine code
Multiple IRs
Source function
JS
Interpreter
Ignition
Baseline compiler
Sparkplug
Mid-tier compiler
Maglev
Top-tier compiler
TurboFan
WA
Baseline compiler
Liftoff
Top-tier compiler
TurboFan

The First Trade-Off

Lower **latency**: faster startup ⟷ Higher **throughput**: peak performance

JS	Interpreter *Ignition*	Baseline compiler *Sparkplug*	Mid-tier compiler *Maglev*	Top-tier compiler *TurboFan*
WA		Baseline compiler *Liftoff*		Top-tier compiler *TurboFan*

What is "High Performance"?

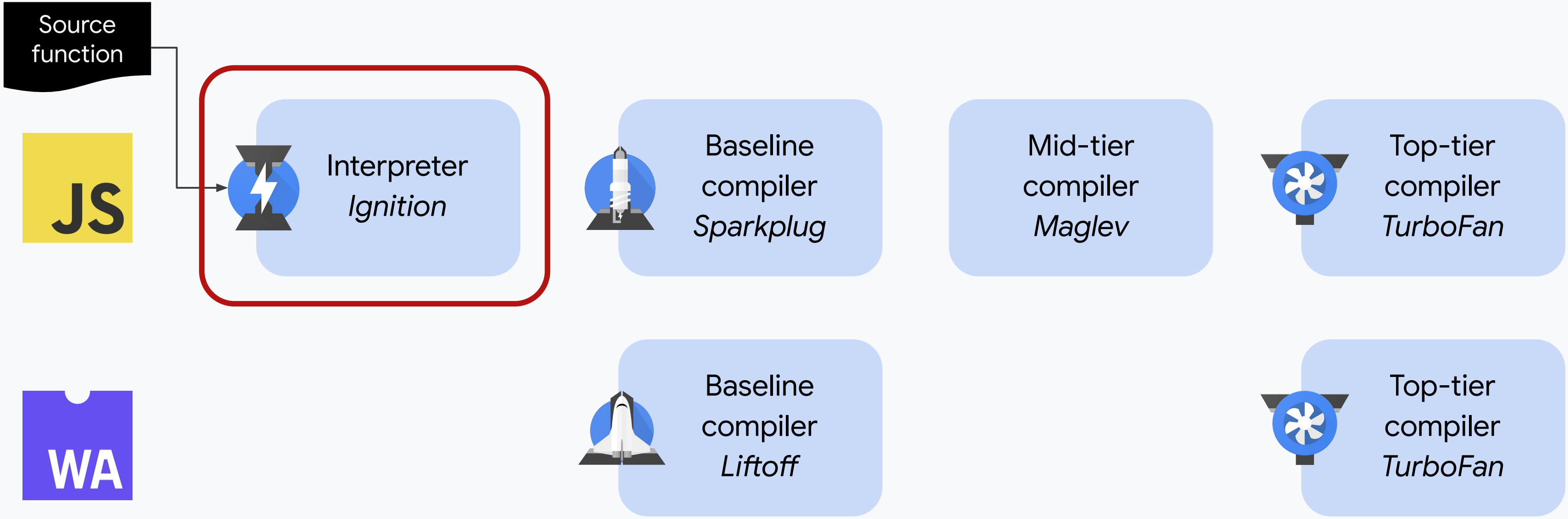
Let's collect ideas:
What do you care
about in the browser?

What is “High Performance”?

- **Peak performance** → execution time of the generated code
- **Latency, startup time** → execution time of the compiler itself
- No **jank** or stutter, e.g., **smooth framerate** → no (long) interruptions of the main thread
- Low **power usage**, especially on mobile devices → minimize total CPU cycles
- **Memory usage,** during execution *and* during compilation → compact data structures
- **Code size**: less memory usage *and* faster execution
- No unnecessary computation → **caching**
- …
- **Code complexity, security**

All of these are important → lots of **trade-offs!**

V8 Overview


Source function
JS
Interpreter
Ignition
Baseline compiler
Sparkplug
Mid-tier compiler
Maglev
Top-tier compiler
TurboFan
WA
Baseline compiler
Liftoff
Top-tier compiler
TurboFan

Bytecode Interpreter

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

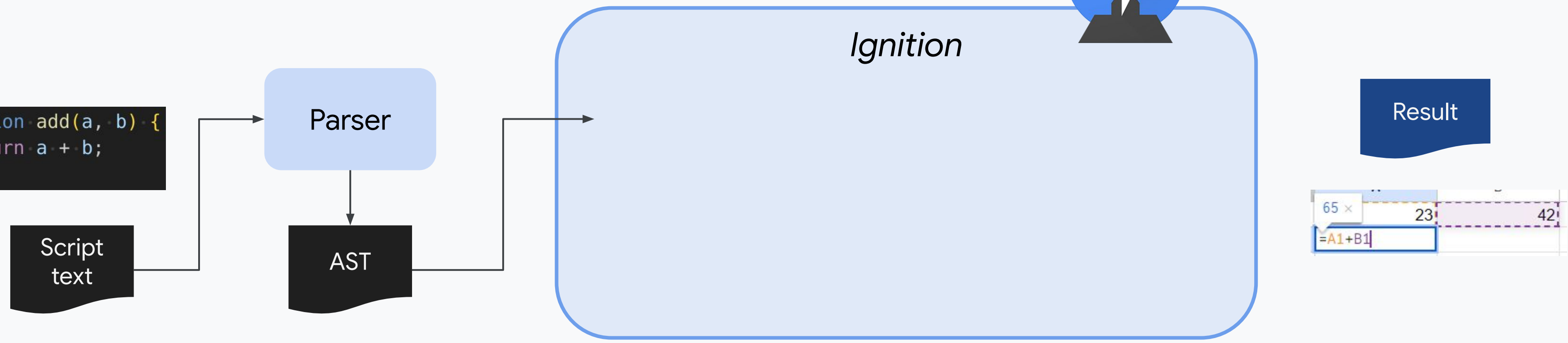
Script text

Ignition

Result

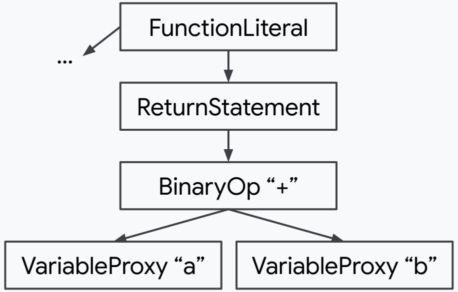
Bytecode Interpreter

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

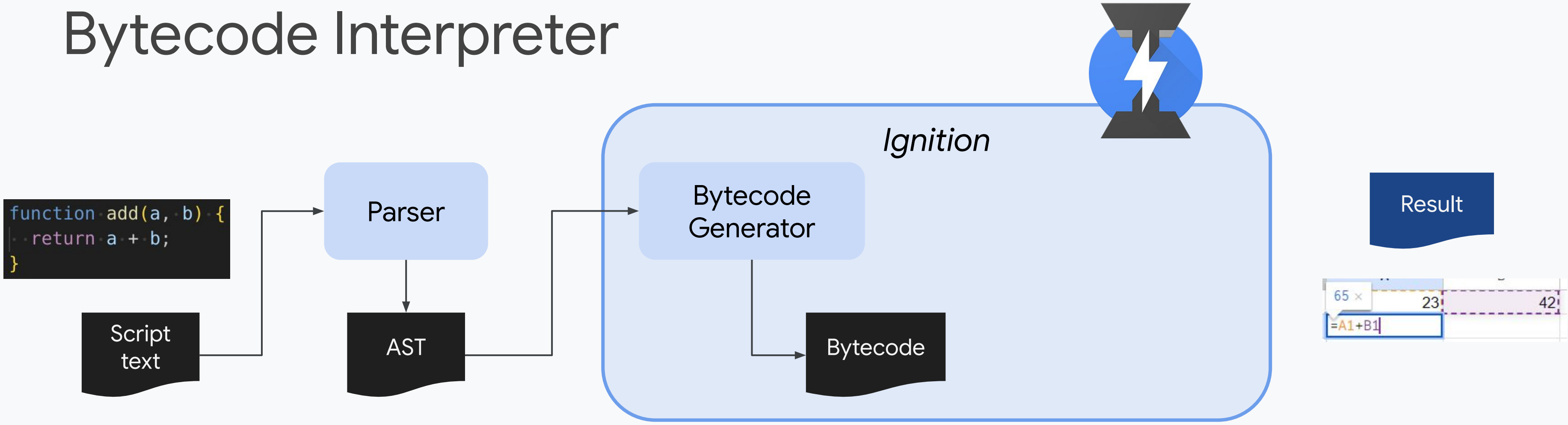



```
FUNC at 96
. NAME "add"
. PARAMS
. . VAR (0x55f153cd4300) (mod
. . VAR (0x55f153cd4380) (mod
. RETURN at 107
. . ADD at 116
. . . VAR PROXY parameter[0]
. . . VAR PROXY parameter[1]
```

Output (simplified)
of --print-ast

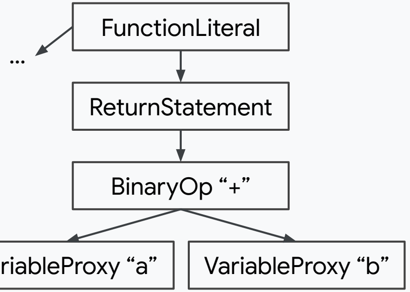


Bytecode Interpreter



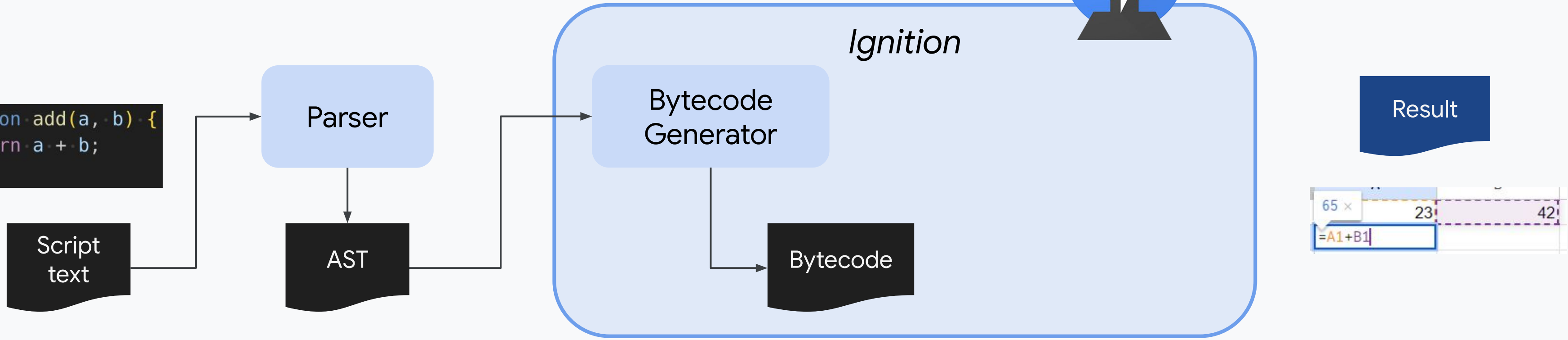

```
FUNC at 96
. NAME "add"
. PARAMS
. . VAR (0x55f153cd4300) (mod
. . VAR (0x55f153cd4380) (mod
. RETURN at 107
. . ADD at 116
. . . VAR PROXY parameter[0]
. . . VAR PROXY parameter[1]
```

Output (simplified) of --print-ast



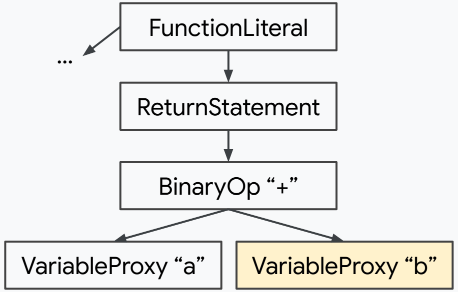
Bytecode Interpreter


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




```
FUNC at 96
. NAME "add"
. PARAMS
. . VAR (0x55f153cd4300) (mod
. . VAR (0x55f153cd4380) (mod
. RETURN at 107
. . ADD at 116
. . . VAR PROXY parameter[0]
. . . VAR PROXY parameter[1]
```

Output (simplified) of --print-ast

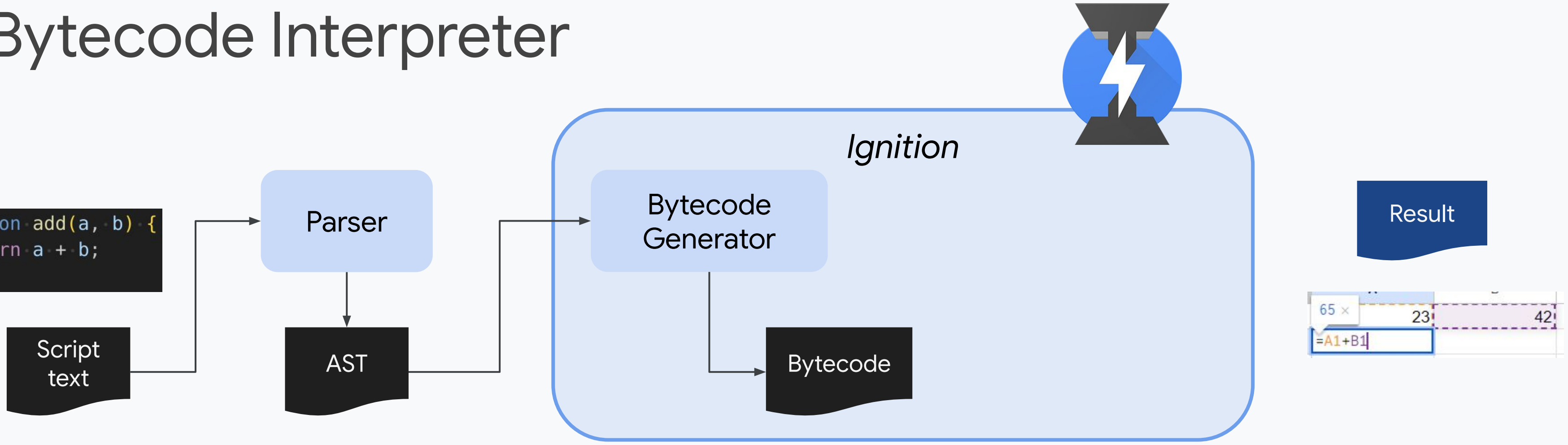



```
0b 04        Ldar a1
```

a0 [a]	23
a1 [b]	42
accumulator	42

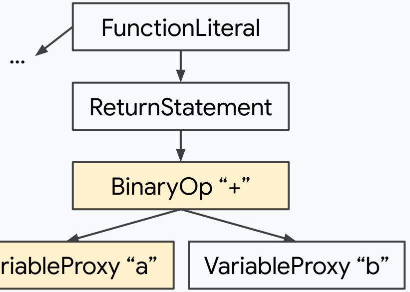
Bytecode Interpreter

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




```
FUNC at 96
. NAME "add"
. PARAMS
. . VAR (0x55f153cd4300) (mod
. . VAR (0x55f153cd4380) (mod
. RETURN at 107
. . ADD at 116
. . . VAR PROXY parameter[0]
. . . VAR PROXY parameter[1]
```

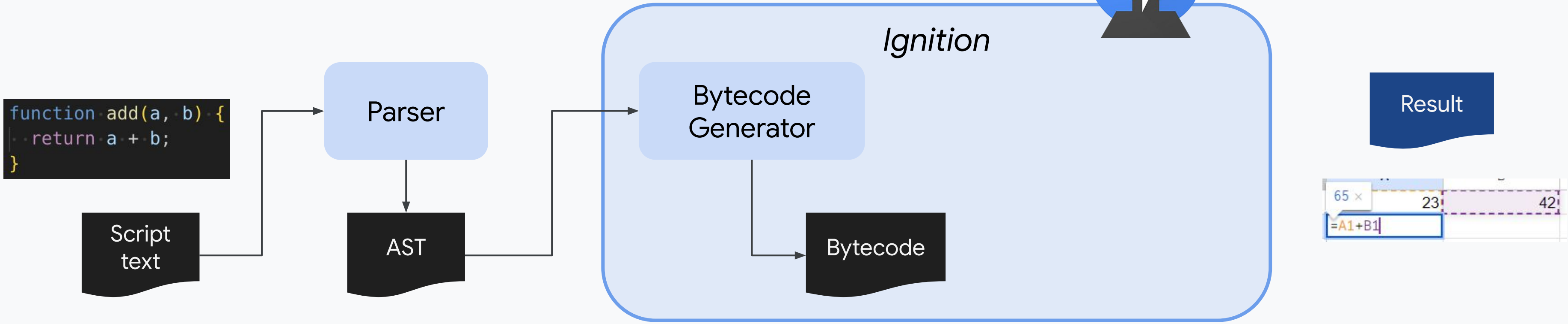
Output (simplified) of --print-ast




```
0b 04       Ldar a1
38 03 00    Add a0, [0]
```

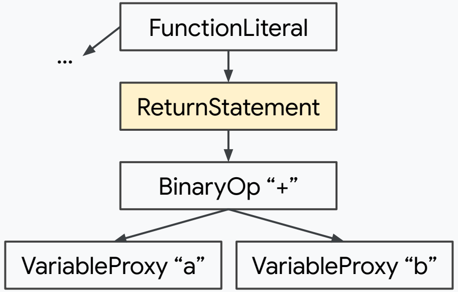
a0 [a]	23
a1 [b]	42
accumulator	65

Bytecode Interpreter



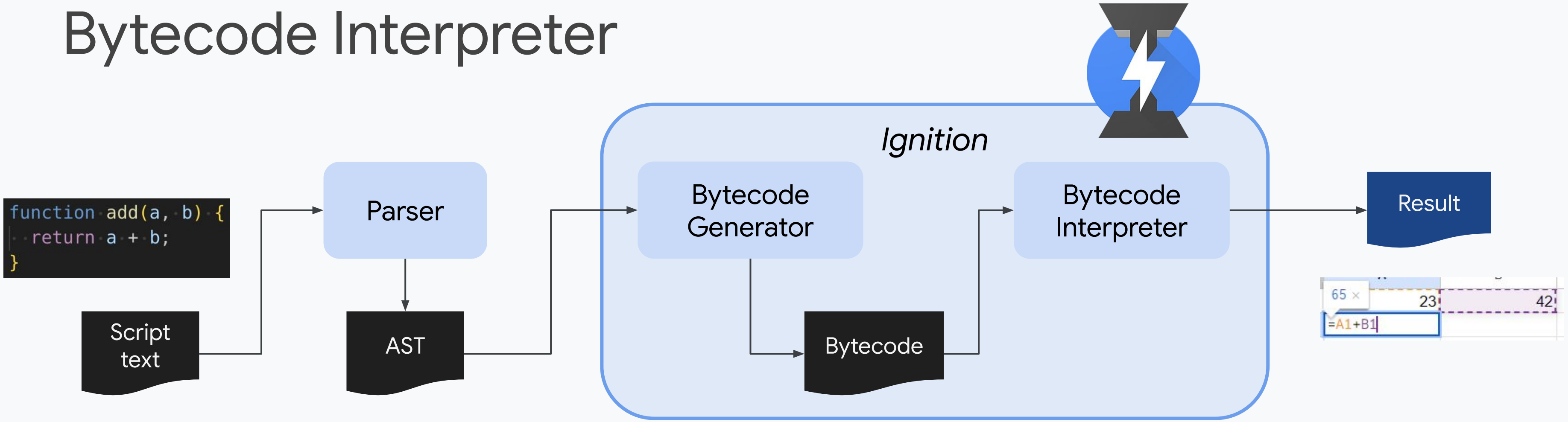

```
FUNC at 96
. NAME "add"
. PARAMS
. . VAR (0x55f153cd4300) (mod
. . VAR (0x55f153cd4380) (mod
. RETURN at 107
. . ADD at 116
. . . VAR PROXY parameter[0]
. . . VAR PROXY parameter[1]
```

Output (simplified) of --print-ast




```
0b 04      Ldar a1
38 03 00   Add a0, [0]
ab         Return
```

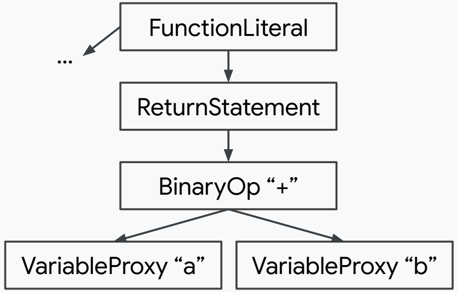
Bytecode Interpreter




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

```
FUNC at 96
. NAME "add"
. PARAMS
. . VAR (0x55f153cd4300) (mod
. . VAR (0x55f153cd4380) (mod
. RETURN at 107
. . ADD at 116
. . . VAR PROXY parameter[0]
. . . VAR PROXY parameter[1]
```

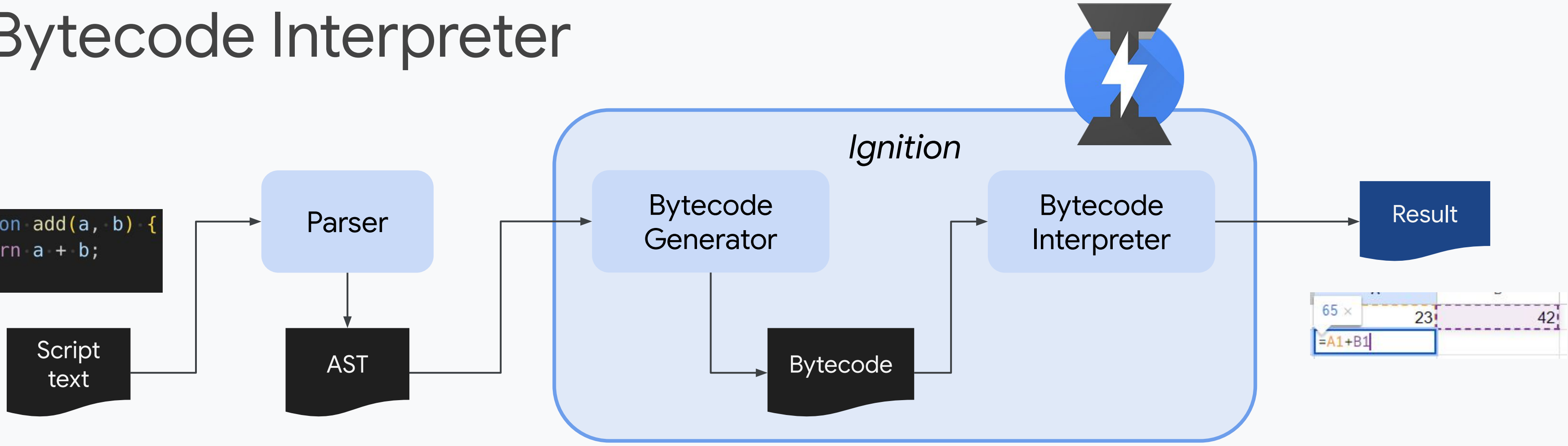
Output (simplified) of --print-ast




```
0b 04       Ldar a1
38 03 00    Add a0, [0]
ab          Return
```

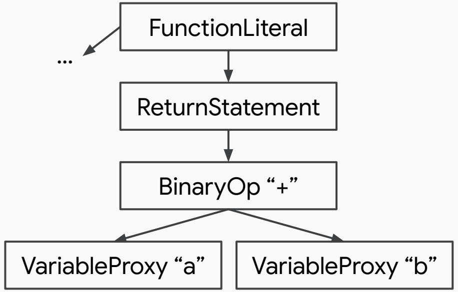
Bytecode Interpreter

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




```
FUNC at 96
. NAME "add"
. PARAMS
. . VAR (0x55f153cd4300) (mod
. . VAR (0x55f153cd4380) (mod
. RETURN at 107
. . ADD at 116
. . . VAR PROXY parameter[0]
. . . VAR PROXY parameter[1]
```

Output (simplified) of --print-ast

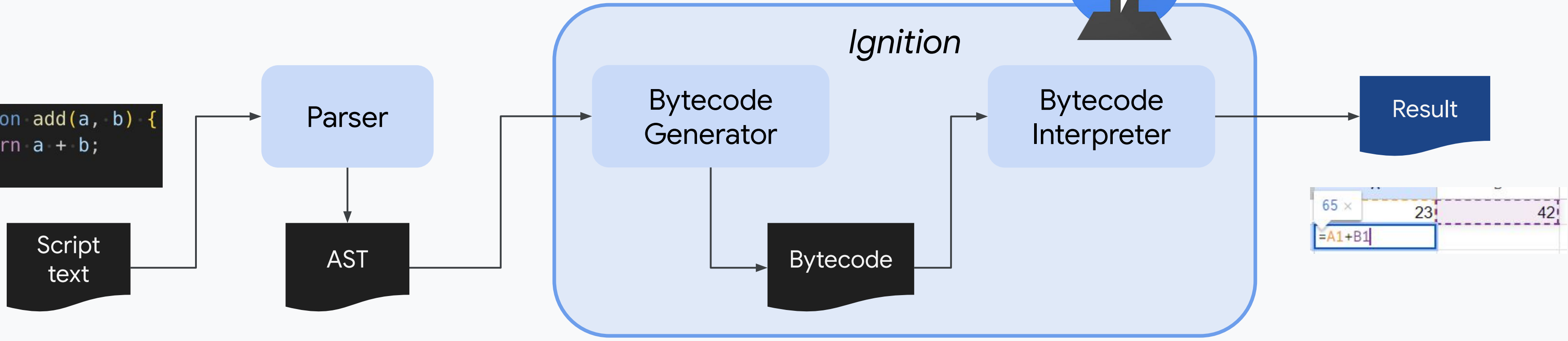



```
0b 04       Ldar a1
38 03 00    Add a0, [0]
ab          Return
```

Why bytecode?

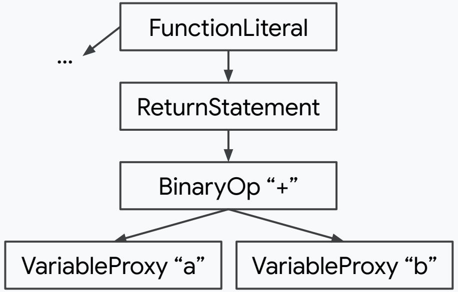
Bytecode Interpreter

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




```
FUNC at 96
. NAME "add"
. PARAMS
. . VAR (0x55f153cd4300) (mod
. . VAR (0x55f153cd4380) (mod
. RETURN at 107
. . ADD at 116
. . . VAR PROXY parameter[0]
. . . VAR PROXY parameter[1]
```

Output (simplified) of --print-ast



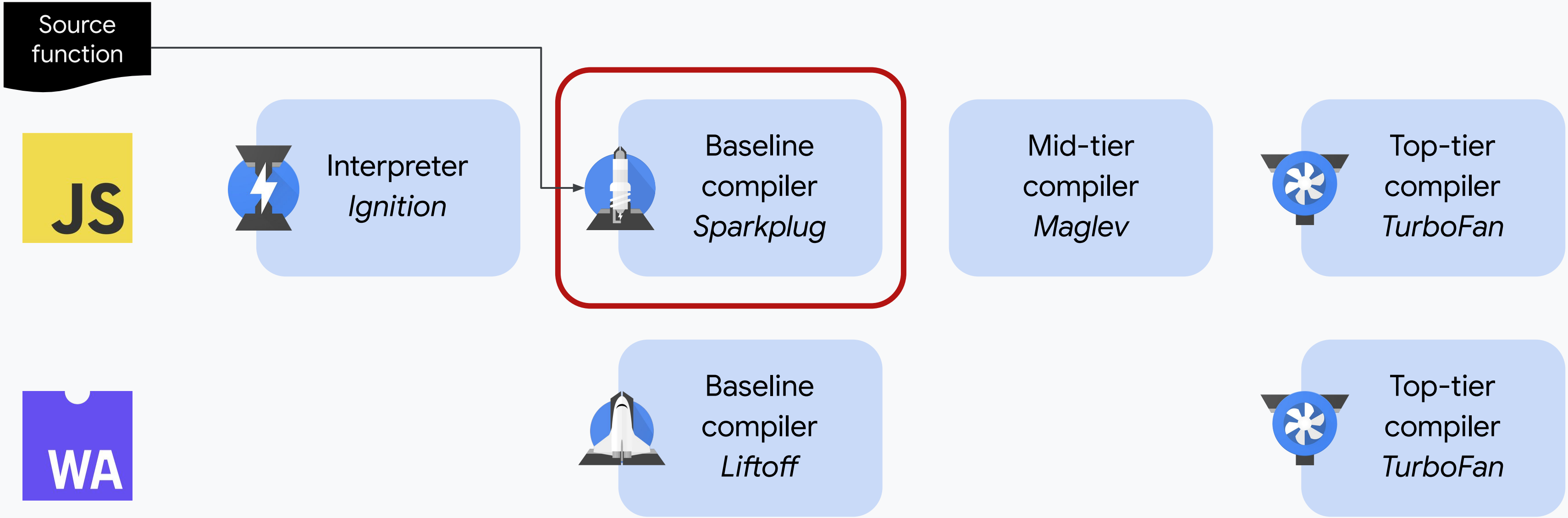

```
0b 04       Ldar a1
38 03 00    Add a0, [0]
ab          Return
```

Why bytecode?

- Parse once, save bytecode
- Faster to interpret
- More compact than AST

→ Faster, memory savings

V8 Overview


Source
function
JS
WA
Interpreter
Ignition
Baseline
compiler
Sparkplug
Mid-tier
compiler
Maglev
Top-tier
compiler
TurboFan
Baseline
compiler
Liftoff
Top-tier
compiler
TurboFan

Sparkplug: A Non-Optimizing JIT Compiler

- Motivation: Eliminate **dispatch overhead** of interpreter
- Compilation itself should be *very* fast: **baseline compiler**

https://v8.dev/blog/sparkplug

Sparkplug: A Non-Optimizing JIT Compiler

- Motivation: Eliminate **dispatch overhead** of interpreter
- Compilation itself should be *very* fast: **baseline compiler**
- No IR, directly generates machine code


```
// The Sparkplug compiler (abridged).
for (; !iterator.done(); iterator.Advance()) {
  VisitSingleBytecode();
}
```

- In essence: Serialization of interpreter execution in native code

https://v8.dev/blog/sparkplug

Sparkplug Compilation

```
0 Ldar a1
2 Sub a0
5 Star0
6 LdaZero
```

(We will just accept V8's calling convention / stack layout without going into details.)

Sparkplug Compilation

```
0 Ldar a1
2 Sub a0
5 Star0
6 LdaZero
```

```
movq rax,[rbp+0x20]
```

Move quad word (64bit move)

Contents of rbp (stack)+0x20

Target register

Sparkplug Compilation

Feedback vector
(will ignore for now)

Builtin function that
implements
JavaScript subtraction

```
0 Ldar a1
2 Sub a0
5 Star0
6 LdaZero
```

```
movq rax,[rbp+0x20]
movq rdx,[rbp+0x18]
xorl rbx,rbx
call 0x55c5be6d0bc0  (Subtract_Baseline)
```

Sparkplug Compilation

```
0 Ldar a1
2 Sub a0
5 Star0
6 LdaZero
```

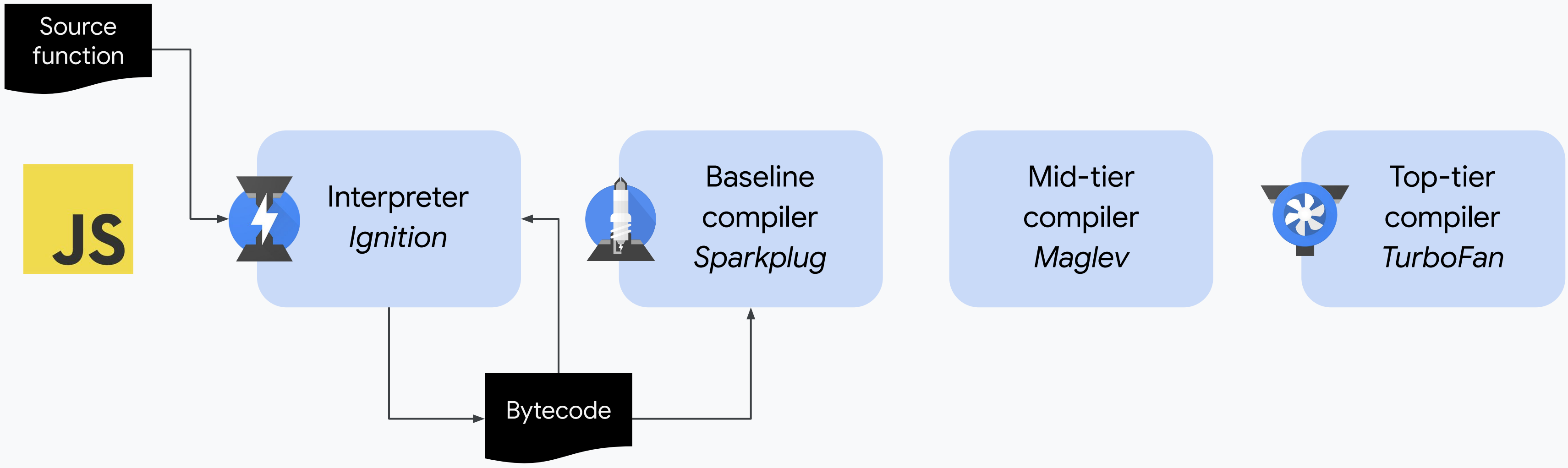
```
movq rax,[rbp+0x20]
movq rdx,[rbp+0x18]
xorl rbx,rbx
call 0x55c5be6d0bc0  (Subtract_Baseline)
movq [rbp-0x30],rax
```

Sparkplug Compilation

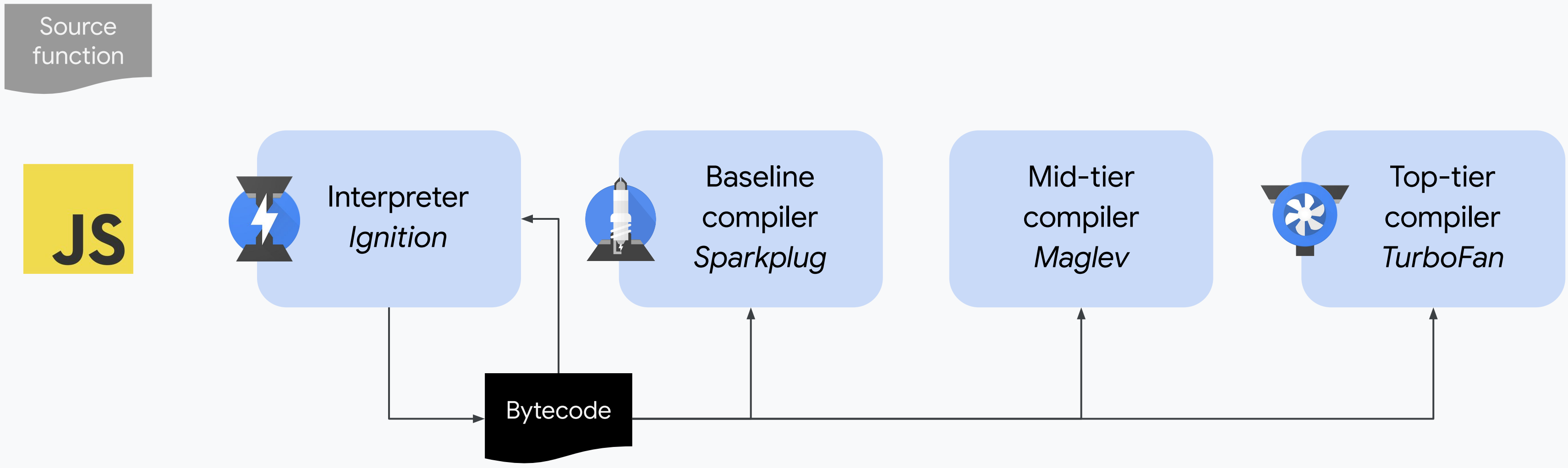
```
0 Ldar a1
2 Sub a0
5 Star0
6 LdaZero
```

```
movq rax,[rbp+0x20]
movq rdx,[rbp+0x18]
xorl rbx,rbx
call 0x55c5be6d0bc0  (Subtract_Baseline)
movq [rbp-0x30],rax
xorl rax,rax
```

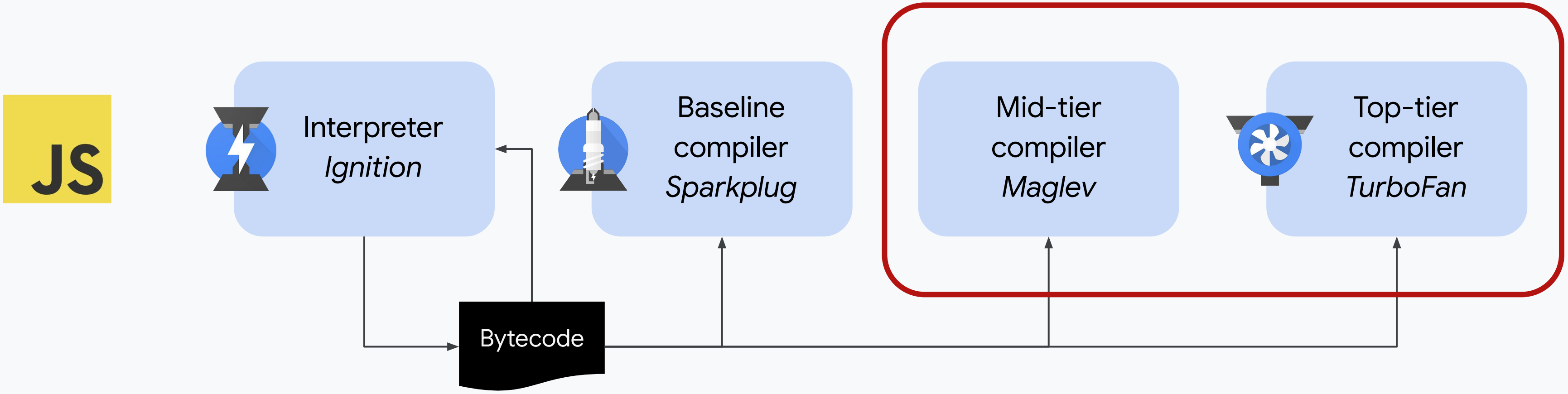
V8 Overview


Source function
JS
Interpreter
Ignition
Baseline compiler
Sparkplug
Mid-tier compiler
Maglev
Top-tier compiler
TurboFan
Bytecode

V8 Overview


Source
function
Interpreter
Ignition
Baseline
compiler
Sparkplug
Mid-tier
compiler
Maglev
Top-tier
compiler
TurboFan
Bytecode

(Speculative) Optimizations



JavaScript “+”

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

add(1, 2);              // 3
```

Integer addition

JavaScript “+”

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

add(1, 2);              // 3
add(1.2, 3.14);         // 4.34
```

Integer addition

Floating point addition

Proprietary

JavaScript “+”

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

add(1, 2);               // 3
add(1.2, 3.14);          // 4.34
add(“hello”, “world”); // “helloworld”
```

Integer addition

Floating point addition

String addition

Google

JavaScript “+”

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

add(1, 2);                // 3
add(1.2, 3.14);           // 4.34
add(“hello”, “world”); // “helloworld”
add(1, true);             // 2
add(“foo”, true);         // “footrue”
```

Integer addition

Floating point addition

String addition

Type coercion

JavaScript “+”

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

add(1, 2);                // 3
add(1.2, 3.14);           // 4.34
add(“hello”, “world”); // “helloworld”
add(1, true);             // 2
add(“foo”, true);         // “footrue”
var bar = {toString:() => “bar”};
add(“foo”, bar);          // “foobar”
```

Integer addition

Floating point addition

String addition

Type coercion

toString() / valueOf()

JavaScript "+" Semantics

12.7.3.1 Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **+** *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lprim* be ToPrimitive(*lval*).
8. ReturnIfAbrupt(*lprim*).
9. Let *rprim* be ToPrimitive(*rval*).
10. ReturnIfAbrupt(*rprim*).
11. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Let *lstr* be ToString(*lprim*).
    b. ReturnIfAbrupt(*lstr*).
    c. Let *rstr* be ToString(*rprim*).
    d. ReturnIfAbrupt(*rstr*).
    e. Return the String that is the result of concatenating *lstr* and *rstr*.
12. Let *lnum* be ToNumber(*lprim*).
13. ReturnIfAbrupt(*lnum*).
14. Let *rnum* be ToNumber(*rprim*).
15. ReturnIfAbrupt(*rnum*).
16. Return the result of applying the **addition** operation to *lnum* and *rnum*. See the Note below 12.7.5.

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.

JavaScript “+” Semantics

operator +

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lprim* be ToPrimitive(*lval*).
8. ReturnIfAbrupt(*lprim*).
9. Let *rprim* be ToPrimitive(*rval*).
10. ReturnIfAbrupt(*rprim*).
11. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Let *lstr* be ToString(*lprim*).
    b. ReturnIfAbrupt(*lstr*).
    c. Let *rstr* be ToString(*rprim*).
    d. ReturnIfAbrupt(*rstr*).
    e. Return the String that is the result of concatenating *lstr* and *rstr*.
12. Let *lnum* be ToNumber(*lprim*).
13. ReturnIfAbrupt(*lnum*).
14. Let *rnum* be ToNumber(*rprim*).
15. ReturnIfAbrupt(*rnum*).
16. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.7.5.

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.

ToPrimitive

Table 9 — ToPrimitive Conversions

Input Type	Result
Completion Record	If *input* is an abrupt completion, return *input*. Otherwise return ToPrimitive(*input*.[[value]]) also passing the optional hint *PreferredType*.
Undefined	Return *input*.
Null	Return *input*.
Boolean	Return *input*.
Number	Return *input*.
String	Return *input*.
Symbol	Return *input*.
Object	Perform the steps following this table.

When Type(*input*) is Object, the following steps are taken:

1. If *PreferredType* was not passed, let *hint* be `"default"`.
2. Else if *PreferredType* is hint String, let *hint* be `"string"`.
3. Else *PreferredType* is hint Number, let *hint* be `"number"`.
4. Let *exoticToPrim* be GetMethod(*input*, @@toPrimitive).
5. ReturnIfAbrupt(*exoticToPrim*).
6. If *exoticToPrim* is not **undefined**, then
    a. Let *result* be Call(*exoticToPrim*, *input*, «*hint*»).
    b. ReturnIfAbrupt(*result*).
    c. If Type(*result*) is not Object, return *result*.
    d. Throw a **TypeError** exception.
7. If *hint* is `"default"`, let *hint* be `"number"`.
8. Return OrdinaryToPrimitive(*input*,*hint*).

When the abstract operation OrdinaryToPrimitive is called with arguments *O* and *hint*, the following steps are taken:

1. Assert: Type(*O*) is Object
2. Assert: Type(*hint*) is String and its value is either `"string"` or `"number"`.
3. If *hint* is `"string"`, then
    a. Let *methodNames* be «`"toString"`, `"valueOf"`».
4. Else,
    a. Let *methodNames* be «`"valueOf"`, `"toString"`».
5. For each *name* in *methodNames* in List order, do
    a. Let *method* be Get(*O*, *name*).
    b. ReturnIfAbrupt(*method*).
    c. If IsCallable(*method*) is **true**, then
        i. Let *result* be Call(*method*, *O*).
        ii. ReturnIfAbrupt(*result*).
        iii. If Type(*result*) is not Object, return *result*.
6. Throw a **TypeError** exception.

NOTE When ToPrimitive is called with no hint, then it generally behaves as if the hint were Number. However, objects may over-ride this behaviour by defining a @@toPrimitive method. Of the objects defined in this specification only Date objects (see 20.3.4.45) and Symbol objects (see 19.4.3.4) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

JavaScript "+" Semantics

operator +

2. Let *rval* be GetValue(*rref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lprim* be ToPrimitive(*lval*).
8. ReturnIfAbrupt(*lprim*).
9. Let *rprim* be ToPrimitive(*rval*).
10. ReturnIfAbrupt(*rprim*).
11. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Let *lstr* be ToString(*lprim*).
    b. ReturnIfAbrupt(*lstr*).
    c. Let *rstr* be ToString(*rprim*).
    d. ReturnIfAbrupt(*rstr*).
    e. Return the String that is the result of concatenating *lstr* and *rstr*.
12. Let *lnum* be ToNumber(*lprim*).
13. ReturnIfAbrupt(*lnum*).
14. Let *rnum* be ToNumber(*rprim*).
15. ReturnIfAbrupt(*rnum*).
16. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.7.5.

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.

ToString

Record	[[value]]).
Undefined	Return `"undefined"`.
Null	Return `"null"`.
Boolean	If *argument* is **true**, return `"true"`. If *argument* is **false**, return `"false"`.
Number	See 7.1.12.1.
String	Return *argument*.
Symbol	Throw a **TypeError** exception.
Object	Apply the following steps: 1. Let *primValue* be ToPrimitive(*argument*, hint String). 2. Return ToString(*primValue*).

7.1.12.1 ToString Applied to the Number Type

The abstract operation ToString converts a Number *m* to String format as follows:

1. If *m* is NaN, return the String `"NaN"`.
2. If *m* is +0 or −0, return the String `"0"`.
3. If *m* is less than zero, return the String concatenation of the String `"-"` and ToString(−*m*).
4. If *m* is +∞, return the String `"Infinity"`.
5. Otherwise, let *n*, *k*, and *s* be integers such that $k \geq 1$, $10^{k-1} \leq s < 10^k$, the Number value for $s \times 10^{n-k}$ is *m*, and *k* is as small as possible. Note that *k* is the number of digits in the decimal representation of *s*, that *s* is not divisible by 10, and that the least significant digit of *s* is not necessarily uniquely determined by these criteria.
6. If $k \leq n \leq 21$, return the String consisting of the code units of the *k* digits of the decimal representation of *s* (in order, with no leading zeroes), followed by $n-k$ occurrences of the code unit 0x0030 (DIGIT ZERO).
7. If $0 < n \leq 21$, return the String consisting of the code units of the most significant *n* digits of the decimal representation of *s*, followed by the code unit 0x002E (FULL STOP), followed by the code units of the remaining $k-n$ digits of the decimal representation of *s*.
8. If $-6 < n \leq 0$, return the String consisting of the code unit 0x0030 (DIGIT ZERO), followed by the code unit 0x002E (FULL STOP), followed by $-n$ occurrences of the code unit 0x0030 (DIGIT ZERO), followed by the code units of the *k* digits of the decimal representation of *s*.
9. Otherwise, if $k = 1$, return the String consisting of the code unit of the single digit of *s*, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether $n-1$ is positive or negative, followed by the code units of the decimal representation of the integer abs($n-1$) (with no leading zeroes).
10. Return the String consisting of the code units of the most significant digit of the decimal representation of *s*, followed by code unit 0x002E (FULL STOP), followed by the code units of the remaining $k-1$ digits of the decimal representation of *s*, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether $n-1$ is positive or negative, followed by the code units of the decimal representation of the integer abs($n-1$) (with no leading zeroes).

ToPrimitive

Table 9 — ToPrimitive Conversions

Input Type	Result
Completion Record	If *input* is an abrupt completion, return *input*. Otherwise return ToPrimitive(*input*.[[value]]) also passing the optional hint *PreferredType*.
Undefined	Return *input*.
Null	Return *input*.
Boolean	Return *input*.
Number	Return *input*.
String	Return *input*.
Symbol	Return *input*.
Object	Perform the steps following this table.

When Type(*input*) is Object, the following steps are taken:

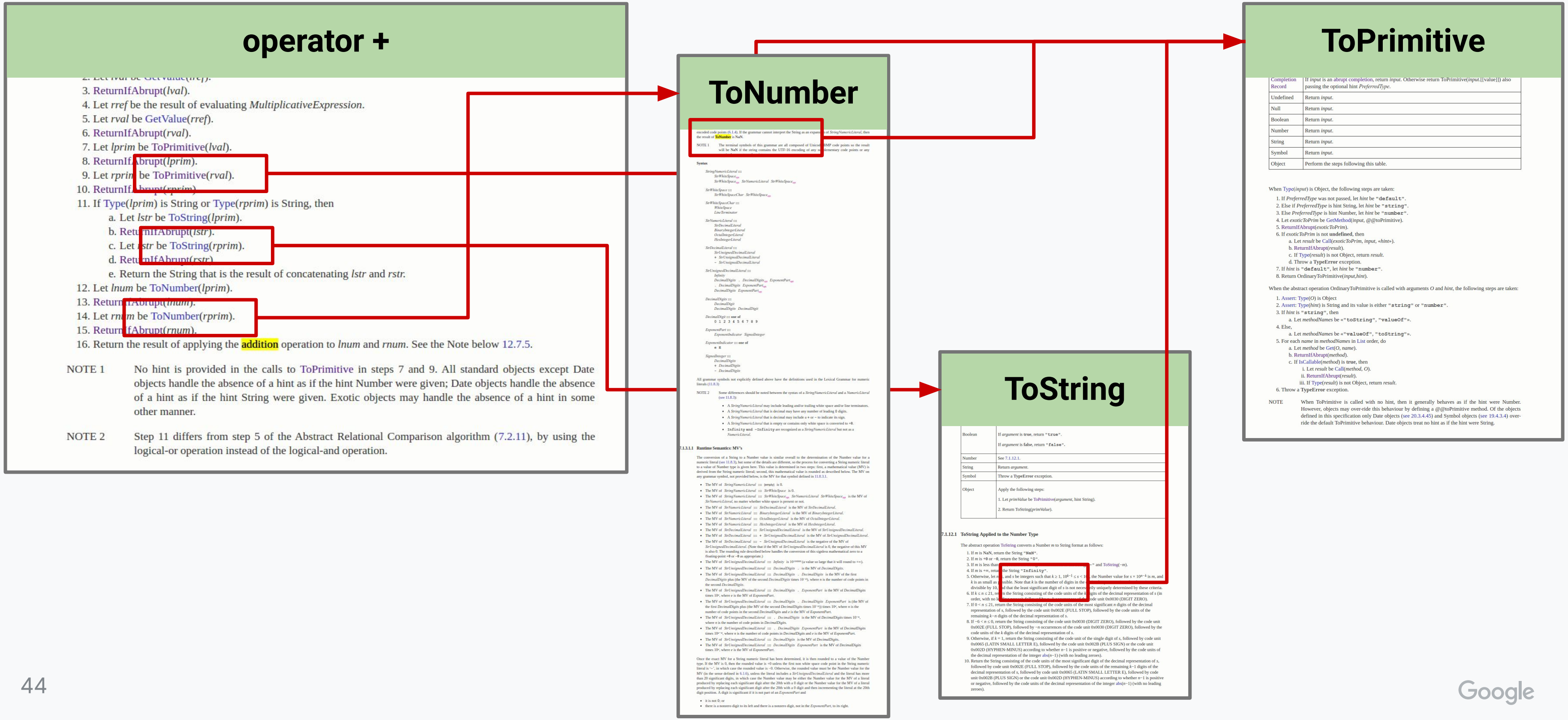
1. If *PreferredType* was not passed, let *hint* be `"default"`.
2. Else if *PreferredType* is hint String, let *hint* be `"string"`.
3. Else *PreferredType* is hint Number, let *hint* be `"number"`.
4. Let *exoticToPrim* be GetMethod(*input*, @@toPrimitive).
5. ReturnIfAbrupt(*exoticToPrim*).
6. If *exoticToPrim* is not **undefined**, then
    a. Let *result* be Call(*exoticToPrim*, *input*, «*hint*»).
    b. ReturnIfAbrupt(*result*).
    c. If Type(*result*) is not Object, return *result*.
    d. Throw a **TypeError** exception.
7. If *hint* is `"default"`, let *hint* be `"number"`.
8. Return OrdinaryToPrimitive(*input*,*hint*).

When the abstract operation OrdinaryToPrimitive is called with arguments *O* and *hint*, the following steps are taken:

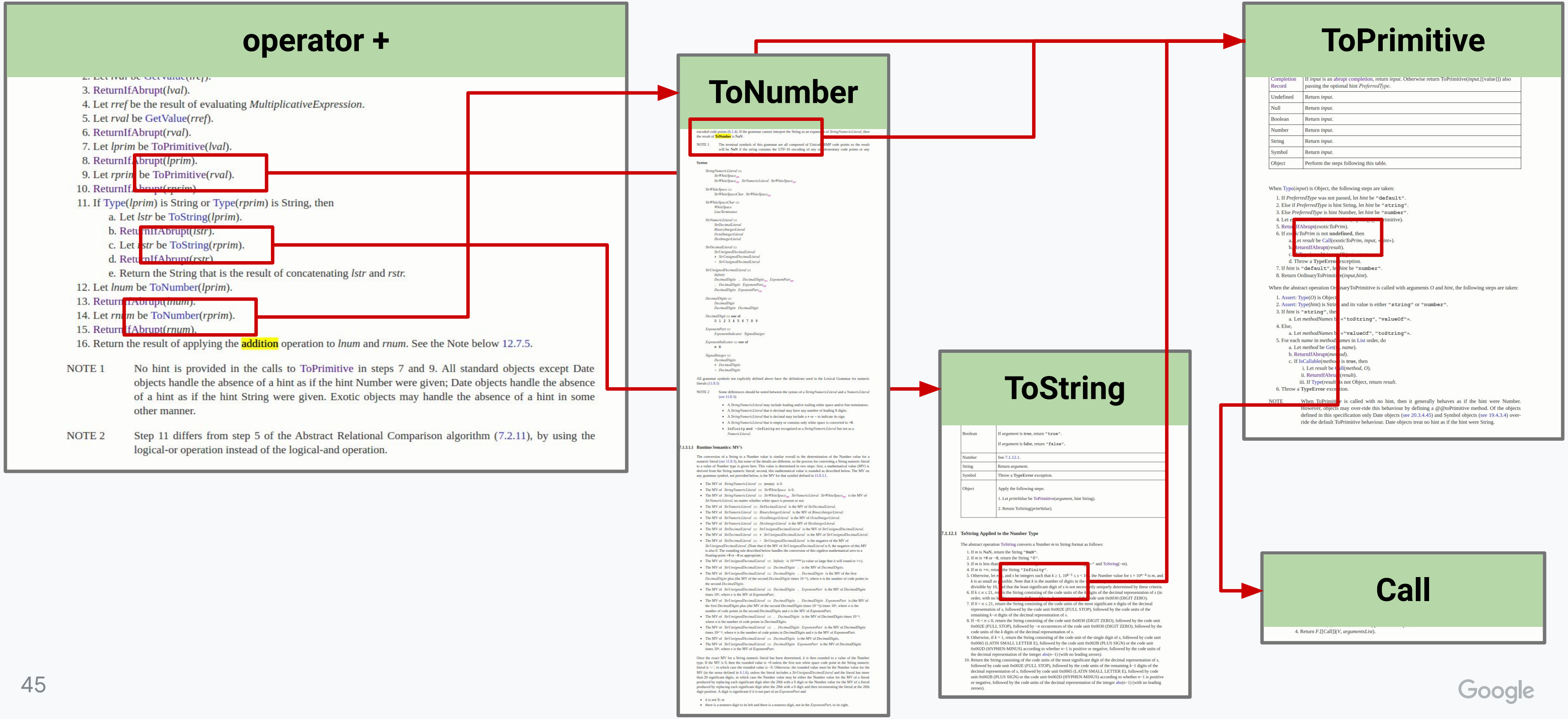
1. Assert: Type(*O*) is Object
2. Assert: Type(*hint*) is String and its value is either `"string"` or `"number"`.
3. If *hint* is `"string"`, then
    a. Let *methodNames* be «`"toString"`, `"valueOf"`».
4. Else,
    a. Let *methodNames* be «`"valueOf"`, `"toString"`».
5. For each *name* in *methodNames* in List order, do
    a. Let *method* be Get(*O*, *name*).
    b. ReturnIfAbrupt(*method*).
    c. If IsCallable(*method*) is **true**, then
        i. Let *result* be Call(*method*, *O*).
        ii. ReturnIfAbrupt(*result*).
        iii. If Type(*result*) is not Object, return *result*.
6. Throw a **TypeError** exception.

NOTE When ToPrimitive is called with no hint, then it generally behaves as if the hint were Number. However, objects may over-ride this behaviour by defining a @@toPrimitive method. Of the objects defined in this specification only Date objects (see 20.3.4.45) and Symbol objects (see 19.4.3.4) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

JavaScript "+" Semantics

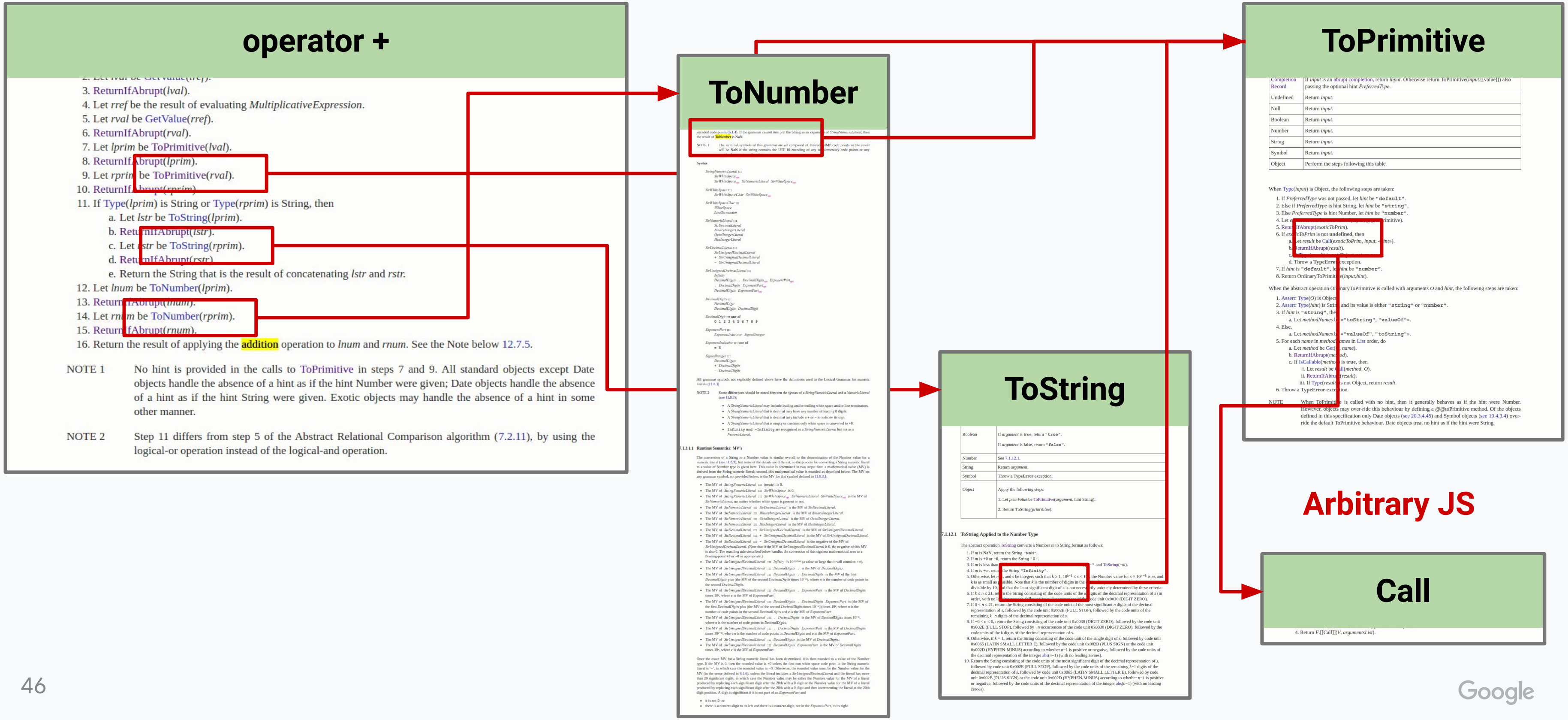


JavaScript “+” Semantics



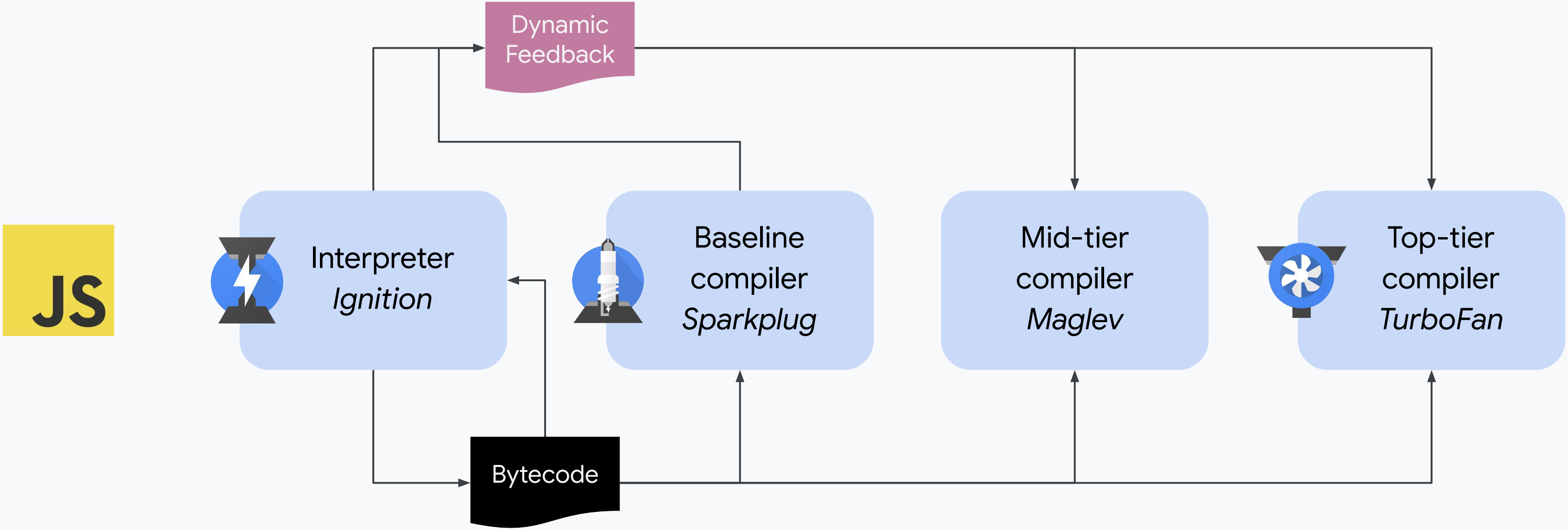
Google

JavaScript “+” Semantics



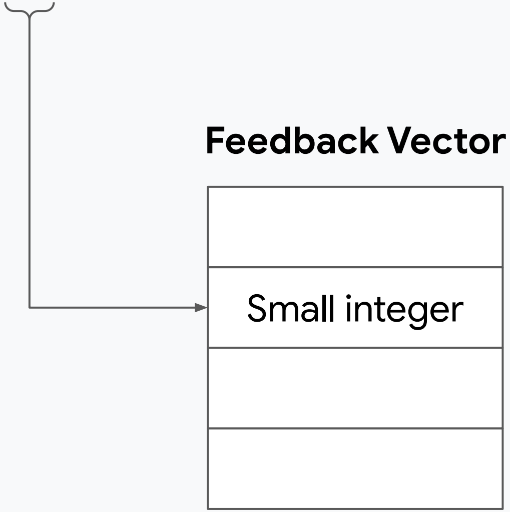
Google

Speculative Optimizations



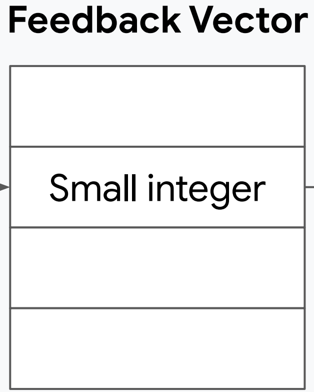
Type Feedback

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

add(1, 2);
```



Type Feedback

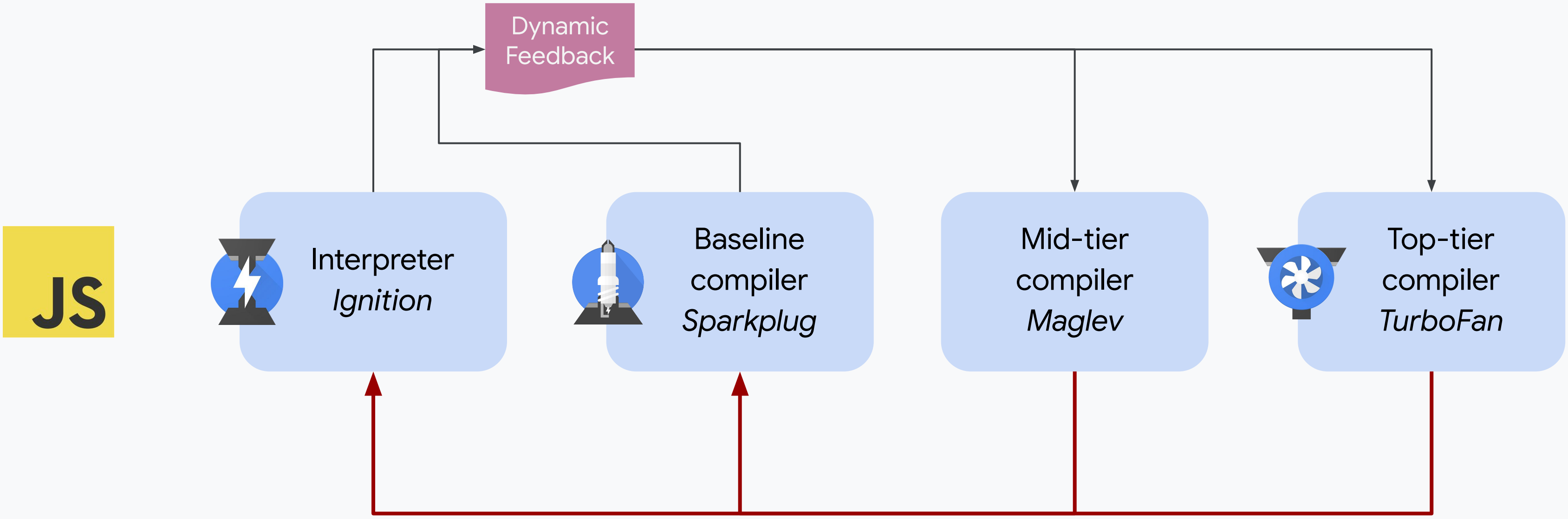
```
function add(a, b) {
  return a + b;
}

add(1, 2);
```



TurboFan generated code:

```
...
# Check that first argument is a Smi.
movq rcx,[rbp+0x18]
testb rcx,0x1
jnz <deopt>
# Convert from Smi to 32-bit word.
movq rdi,rcx
sarl rdi, 1
# Check that second argument is a Smi.
movq r8,[rbp+0x20]
testb r8,0x1
jnz <deopt>
# Convert from Smi to 32-bit word.
movq r9,r8
sarl r9, 1
# Addition, check for overflow
addl r9,rdi
jo <deopt>
...
```

Speculative Optimizations

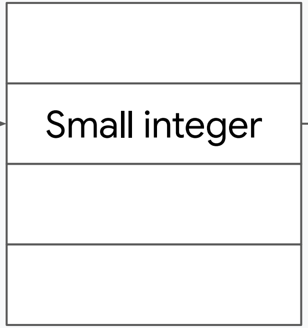


Deopt: Return to more general code (typically in lower tier), because some speculative assumption was violated.

Type Feedback

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

add(1, 2);
```

Feedback Vector

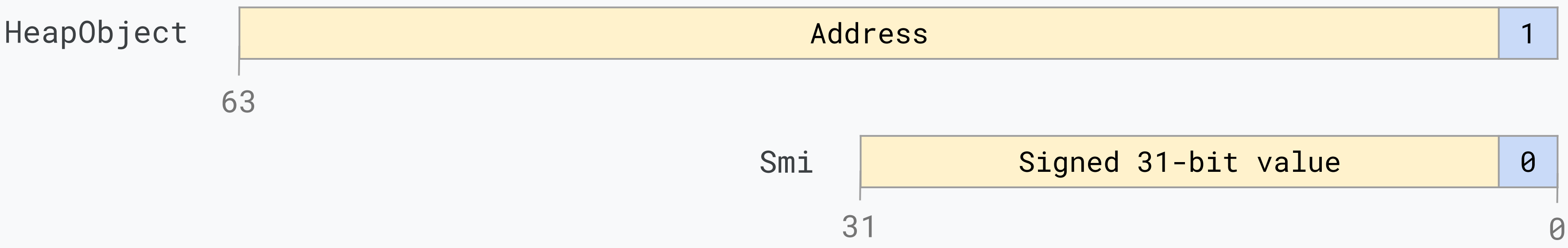


TurboFan generated code:

```
...
# Check that first argument is a Smi.
movq rcx,[rbp+0x18]
testb rcx,0x1
jnz <deopt>
# Convert from Smi to 32-bit word.
movq rdi,rcx
sarl rdi, 1
# Check that second argument is a Smi.
movq r8,[rbp+0x20]
testb r8,0x1
jnz <deopt>
# Convert from Smi to 32-bit word.
movq r9,r8
sarl r9, 1
# Addition, check for overflow
addl r9,rdi
jo <deopt>
...
```

Optimized Representations

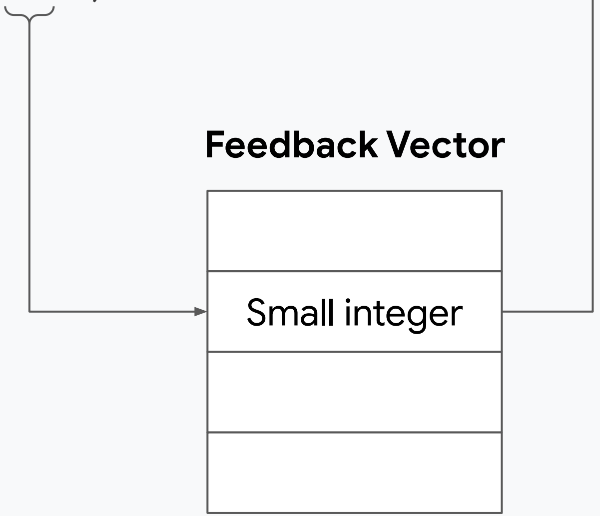
- Heap object / allocation for every number would be **slow** and **wasteful**
- **Unbox** them, i.e., store value directly instead of pointer to value
- Distinguish from pointers / objects via **tag bit**:



Type Feedback

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

add(1, 2);
```

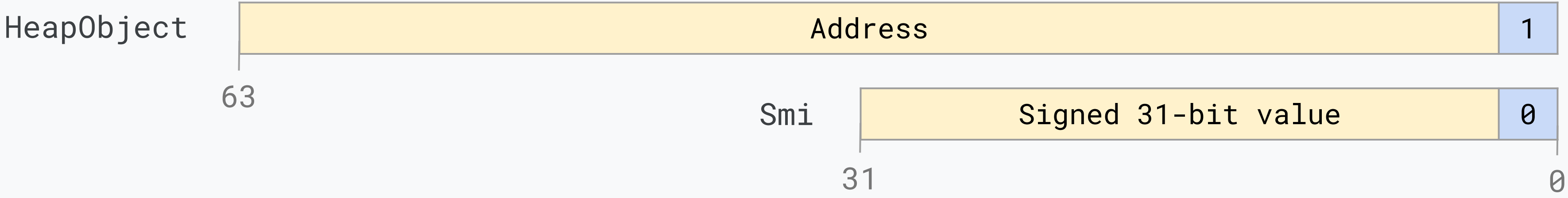
Feedback Vector



TurboFan generated code:

```
...
# Check that first argument is a Smi.
movq rcx,[rbp+0x18]
testb rcx,0x1
jnz <deopt>
# Convert from Smi to 32-bit word.
movq rdi,rcx
sarl rdi, 1
# Check that second argument is a Smi.
movq r8,[rbp+0x20]
testb r8,0x1
jnz <deopt>
# Convert from Smi to 32-bit word.
movq r9,r8
sarl r9, 1
# Addition, check for overflow
addl r9,rdi
jo <deopt>
...
```

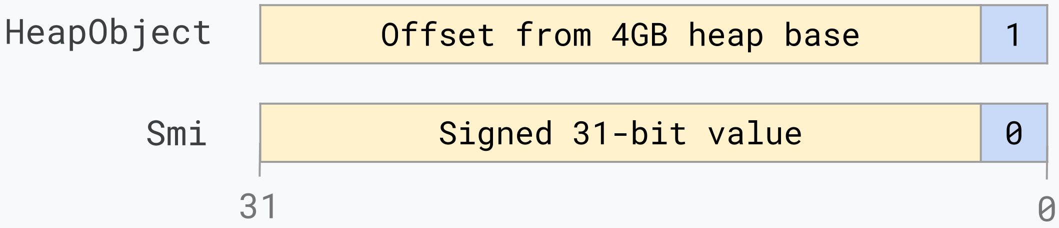
Pointer Compression



- 64-bit pointers to HeapObjects are also wasteful (much fewer than 2^32 objects) → 4 byte “pointers” (offsets)

https://v8.dev/blog/pointer-compression

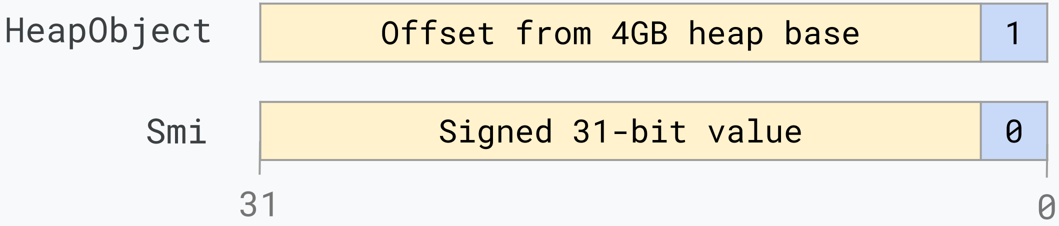
Pointer Compression



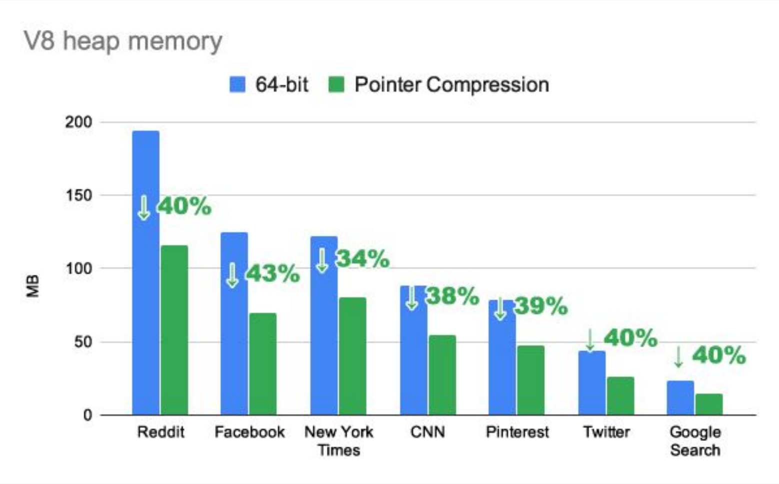
- 64-bit pointers to HeapObjects are also wasteful (much fewer than 2^32 objects)
→ 4 byte “pointers” (offsets)

https://v8.dev/blog/pointer-compression

Pointer Compression



- 64-bit pointers to HeapObjects are also wasteful (much fewer than 2^32 objects) → 4 byte “pointers” (offsets)
- Quite large memory savings (browsing on Windows 10):



https://v8.dev/blog/pointer-compression

V8 Overview

JS

Interpreter
Ignition

Baseline
compiler
Sparkplug

Mid-tier
compiler
Maglev

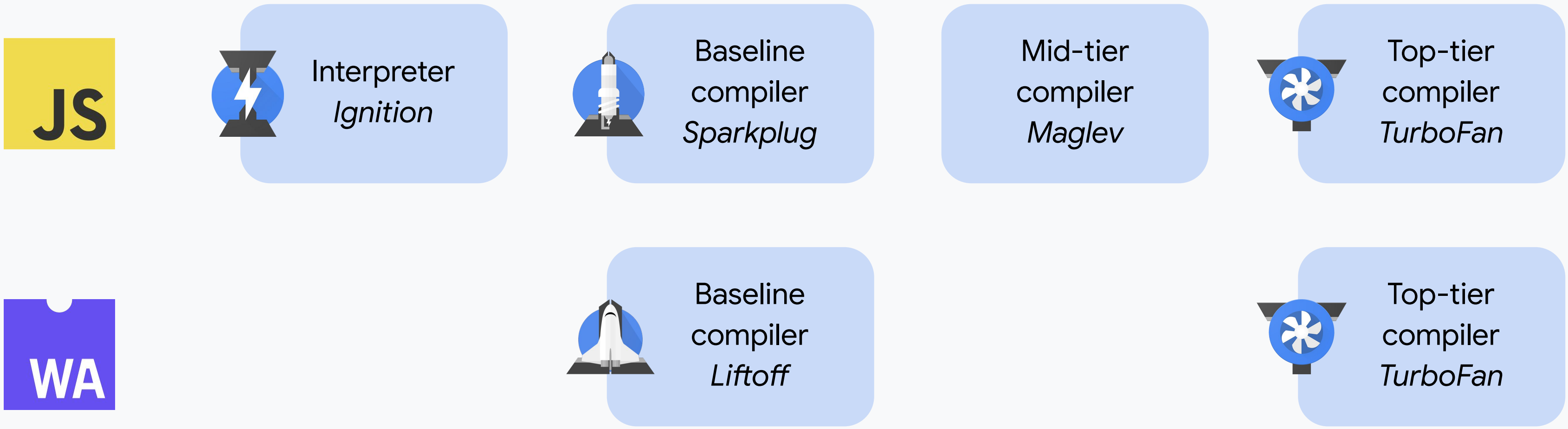
Top-tier
compiler
TurboFan

WA

Baseline
compiler
Liftoff

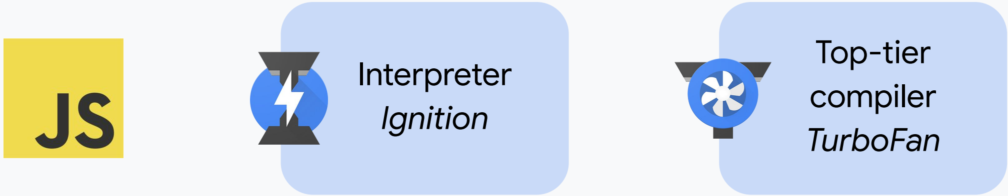
Top-tier
compiler
TurboFan

Do All Tiers Matter?



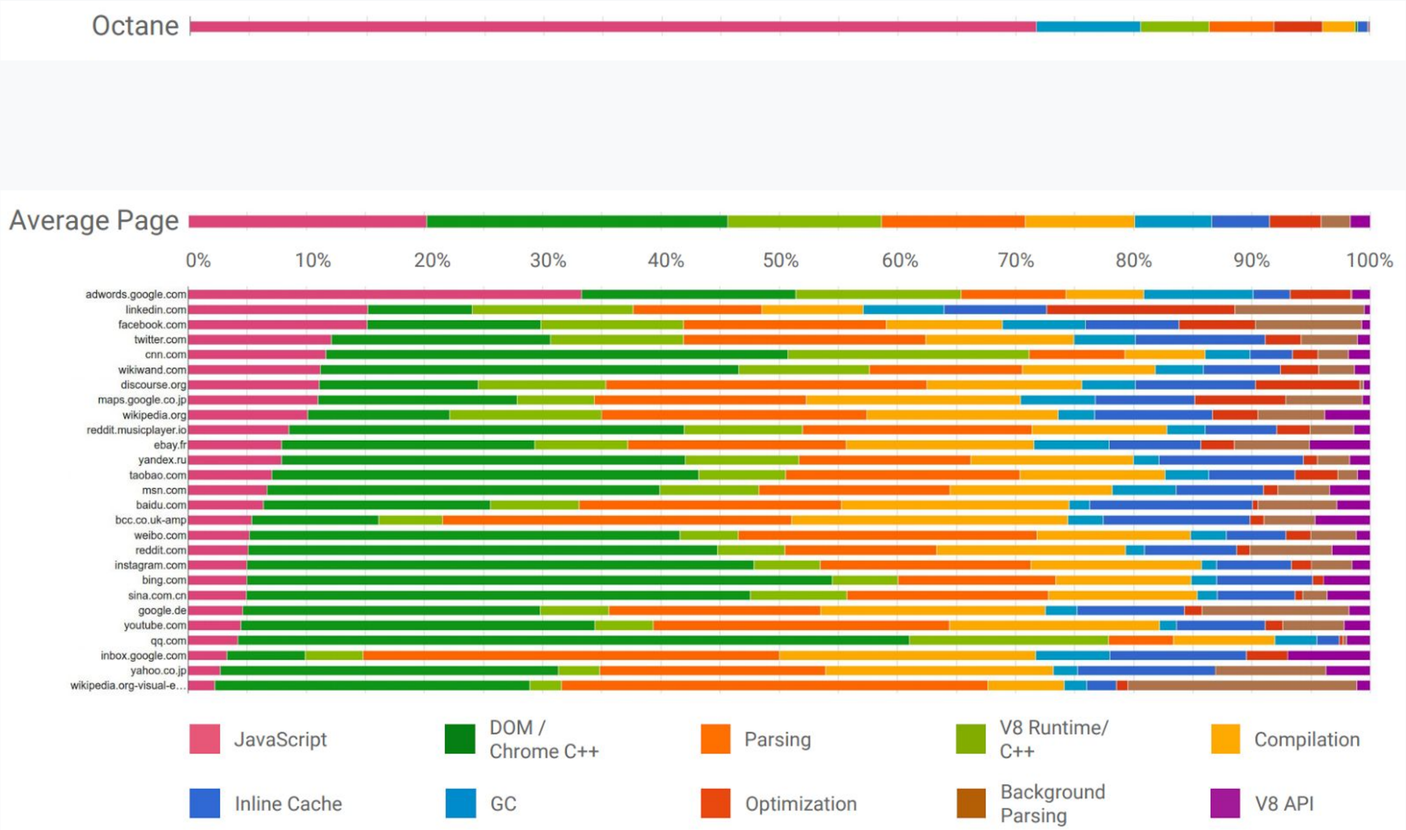
Performance Measurements

- Careful with benchmarks!
- This is what people used to measure: Lots of small, hot JavaScript functions
  → Optimizes for peak performance
- Pre-2020 compilation pipeline:

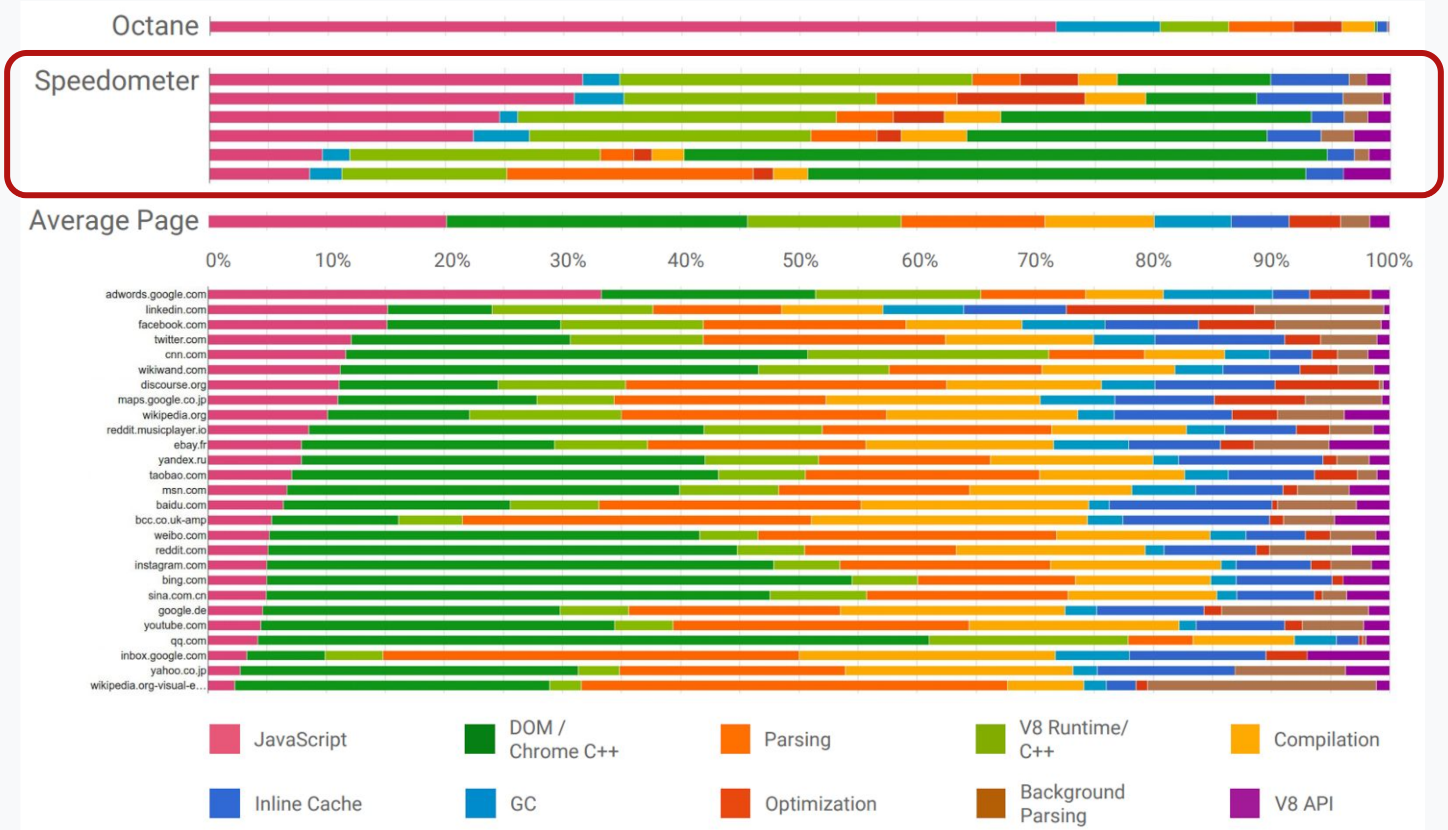


Performance Measurements

Performance Measurements



Performance Measurements



Do All Tiers Matter?

JS

Interpreter
Ignition

Baseline
compiler
Sparkplug

Mid-tier
compiler
Maglev

Top-tier
compiler
TurboFan

Do All Tiers Matter?

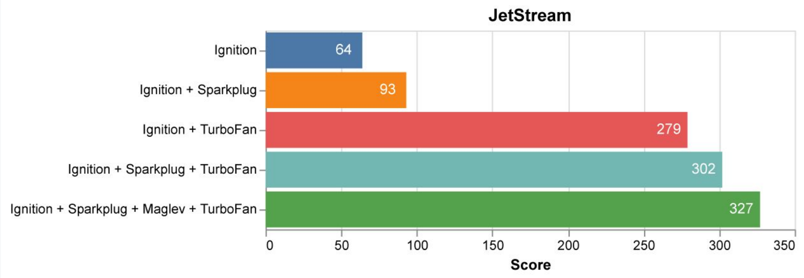
JS

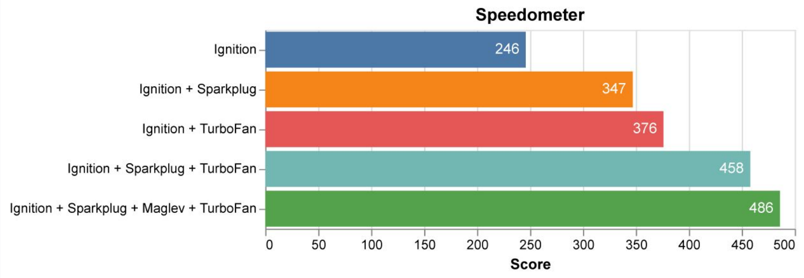
Interpreter
Ignition

Baseline compiler
Sparkplug

Mid-tier compiler
Maglev

Top-tier compiler
TurboFan





Yes!

https://v8.dev/blog/maglev

Maglev: The Latest Addition!

JS

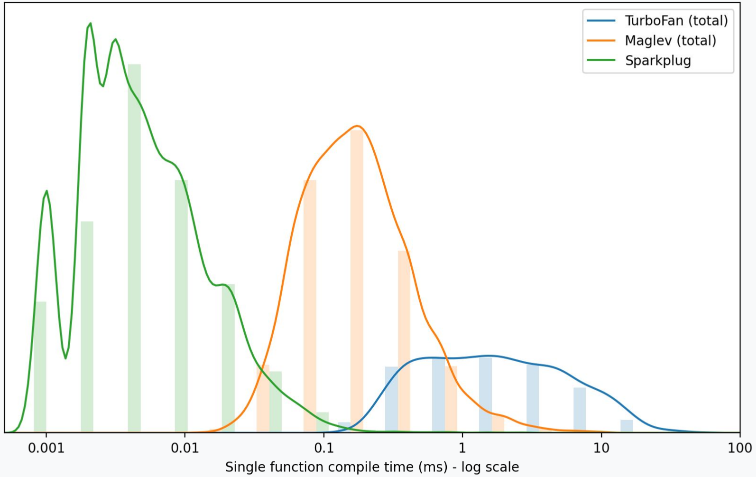
Interpreter
Ignition

Baseline
compiler
Sparkplug

Mid-tier
compiler
Maglev

Top-tier
compiler
TurboFan

- Introduced in 2023
- Sits between Sparkplug and TurboFan
- We can now tier-up later to TurboFan
- Saves total CPU time and thus **power**
  - -3.5% on JetStream
  - -10% on Speedometer



https://v8.dev/blog/maglev

V8 Overview

JS

Interpreter
Ignition

Baseline
compiler
Sparkplug

Mid-tier
compiler
Maglev

Top-tier
compiler
TurboFan

WA

Baseline
compiler
Liftoff

Top-tier
compiler
TurboFan

WebAssembly

- Binary format: compact, fast to parse
- Low-level compilation target, e.g., from C++, Rust, …
- Much less dynamic than JavaScript: static types, no eval

WebAssembly vs. JavaScript

- Binary format: compact, fast to parse
- Low-level compilation target, e.g., from C++, Rust, …
- Much less dynamic than JavaScript: static types, no eval

emscripten

JS

Interpreter
Ignition

Baseline
compiler
Sparkplug

Mid-tier
compiler
Maglev

Top-tier
compiler
TurboFan

WA

Interpreter?

Baseline
compiler
Liftoff

Top-tier
compiler
TurboFan

WebAssembly vs. JavaScript

- Binary format: compact, fast to parse
- Low-level compilation target, e.g., from C++, Rust, …
- Much less dynamic than JavaScript: static types, no eval

emscripten

JS

Interpreter
Ignition

Baseline
compiler
Sparkplug

Mid-tier
compiler
Maglev

Top-tier
compiler
TurboFan

WA

Interpreter?

- Wasm operation much "smaller"
- More dispatch compared to JS

→ Wasm favors compilation

Baseline
compiler
Liftoff

Top-tier
compiler
TurboFan

WebAssembly vs. JavaScript

- Binary format: compact, fast to parse
- Low-level compilation target, e.g., from C++, Rust, …
- Much less dynamic than JavaScript: static types, no eval

emscripten

JS

Interpreter
Ignition

Baseline
compiler
Sparkplug

Mid-tier
compiler
Maglev

Top-tier
compiler
TurboFan

WA

Baseline
compiler
Liftoff

Fewer Optimizations?

Top-tier
compiler
TurboFan

WebAssembly vs. JavaScript

- Binary format: compact, fast to parse
- Low-level compilation target, e.g., from C++, Rust, ...
- Much less dynamic than JavaScript: static types, no eval

emscripten

JS

Interpreter *Ignition*

Baseline compiler *Sparkplug*

Mid-tier compiler *Maglev*

Top-tier compiler *TurboFan*

WA

Baseline compiler *Liftoff*

Fewer Optimizations?

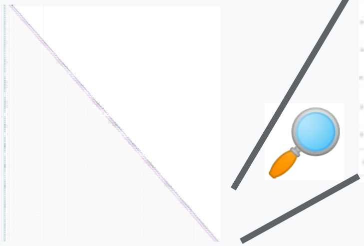
Ahead-of-time compiler (generating Wasm) already Optimized statically.

Top-tier compiler *TurboFan*

Story Time: Huge Wasm Functions

- JetStream 2 tsf-wasm benchmark:
  Liftoff-generated code *faster* than TurboFan code!?

- Interesting WebAssembly bytecode:



```
block $B293
  block $B294
    block $B295
      block $B296
        block $B297
          local.get $16
          i32.load
          br_table $B292 $B297 $B291 $B290 $
```

~300 nested blocks,
branch table in the middle

- Fallback to faster register allocator
  for very large functions

pipeline.cc (@ 4e9d946)

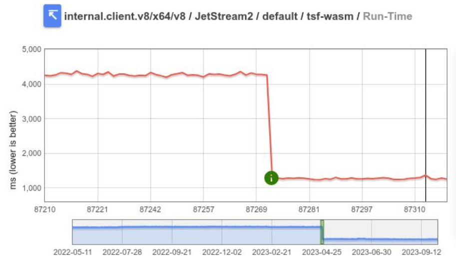
```
// Allocate registers.

// This limit is chosen somewhat arbitrarily, by looking at a few bigger
// WebAssembly programs, and chosing the limit such that functions that take
// >100ms in register allocation are switched to mid-tier.
static int kTopTierVirtualRegistersLimit = 8192;
```

Easy fix, right?

```
3918        static int kTopTierVirtualRegistersLimit = 8192;
     3918   static int kTopTierVirtualRegistersLimit = 16384;
```

- Just use better register allocator for large functions?

→ Did improve runtime by a lot! 😀



- But better register allocator uses more Memory and slows compilation down! 🙁

→ Can we have both?

Issue 9529: Slow TurboFan compilation of Async Julia wasm

Reported by azakai@google.com on Tue, Jul 23, 2019, 2:41 AM GMT+2 Code

Project Member

Version: 7.7.0 (117ddc8f6d026dfef11a61a93467956d9247868c)
OS: Linux
Architecture: x64

This testcase is the Julia language's repl (background: https://github.com/JuliaLang/julia/pull/325
very slowly in TurboFan,

time d8 --no-wasm-tier-up a.out.js

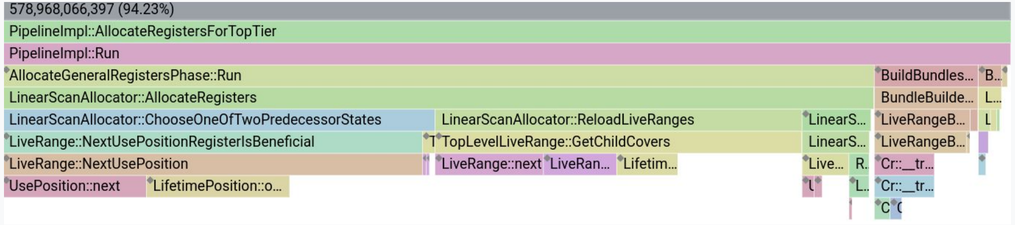
on this testcase takes 6 minutes 49 seconds. In the browser this is noticeable as 100% CPU usage

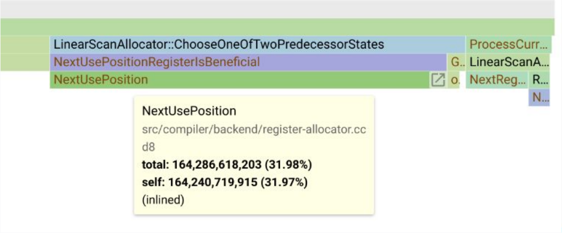
Google

Working on Chrome is a bit like changing the engine of a Formula 1 car… while it's driving.

Speeding Up the Top-Tier Register Allocator

- Tools: Get familiar with profilers (e.g., *Linux perf*), *flamegraphs*, *heaptrack*





- Common optimization themes:
  - Store data inline, avoid temporary allocations!
  - Cache-friendly data structures, avoid linked lists!
  - Micro-optimizations: custom calling convention, statically disable tracing code, …

- We could finally get both: good generated code *and* compile times! 🥳

Complexity → Security?

JS

Interpreter
Ignition

Baseline compiler
Sparkplug

Mid-tier compiler
Maglev

Top-tier compiler
TurboFan

WA

Baseline compiler
Liftoff

Top-tier compiler
TurboFan

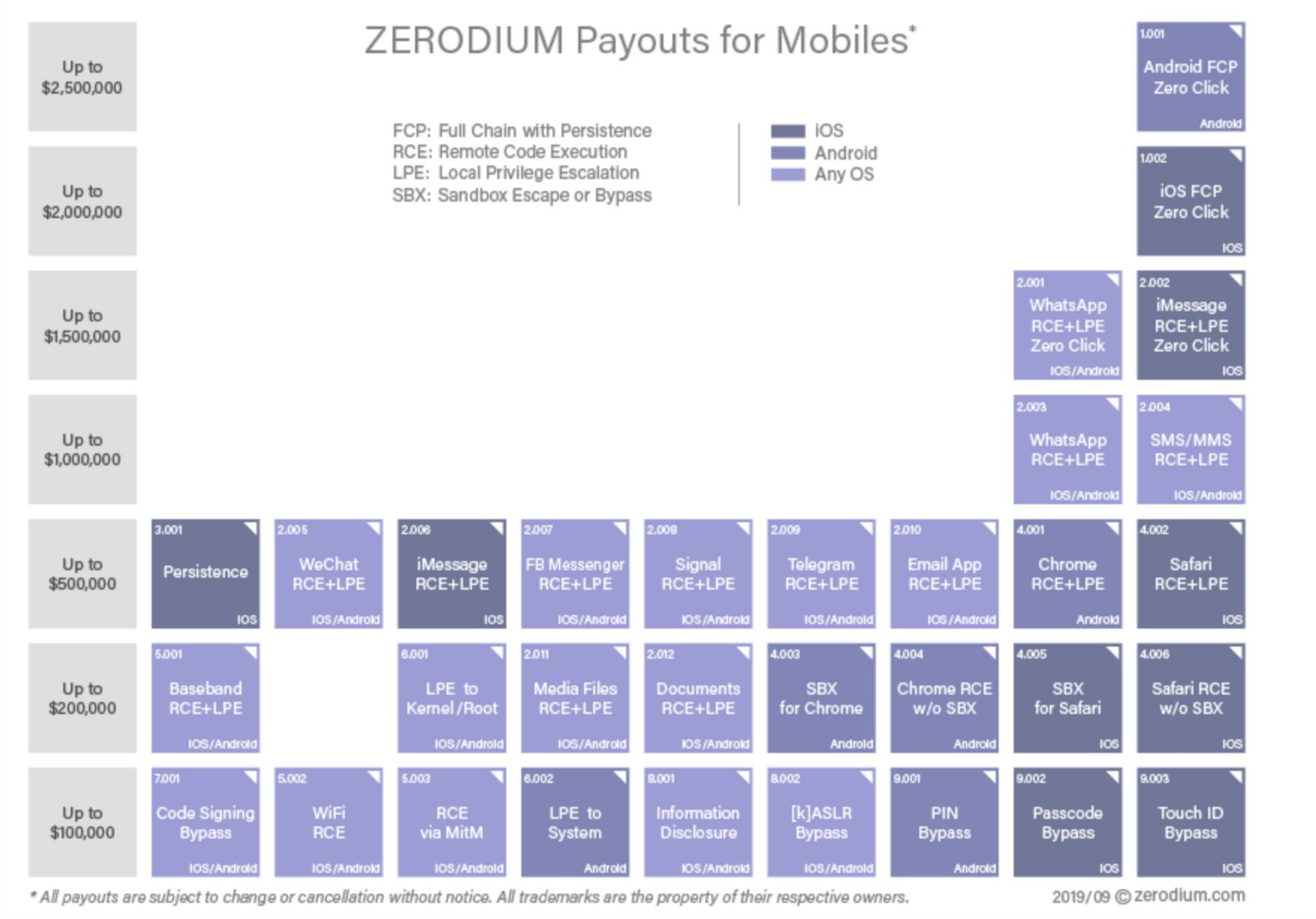
ZERODIUM Payouts for Desktops/Servers*

Windows
macOS
Linux/BSD
Any OS

RCE: Remote Code Execution
LPE: Local Privilege Escalation
SBX: Sandbox Escape or Bypass
VME: Virtual Machine Escape

Up to $1,000,000									1.001 Win RCE Zero Click (Win)
Up to $500,000							3.001 Chrome RCE+LPE (Win)	2.001 Apache RCE (Linux)	2.002 MS IIS RCE (Win)
Up to $250,000						5.001 MS Outlook RCE (Win)	4.001 MS Exchange RCE (Win)	2.003 OpenSSL RCE (Linux)	2.004 PHP RCE (Linux)
Up to $200,000	6.001 VMware ESXi VME	5.002 Thunderbird RCE (Win/Linux)			4.002 Sendmail RCE (Linux)	4.003 Postfix RCE (Linux)	4.004 Dovecot RCE (Linux)	4.005 Exim RCE (Linux)	2.005 nginx RCE (Linux)
Up to $100,000		3.002 Safari RCE+LPE (Mac)	3.003 Edge RCE+LPE (Win)	3.004 Firefox RCE+LPE (Win)	5.003 Word/Excel RCE (Win)	7.001 WordPress RCE (Linux)	7.002 cPanel/WHM RCE (Linux)	7.003 Plesk RCE (Linux)	7.004 Webmin RCE (Linux)
Up to $80,000	6.002 VMware WS VME (Win/Linux)					5.004 Adobe PDF RCE+SBX (Win)	5.005 WinRAR RCE (Win)	5.006 7-Zip RCE (Win)	6.003 Windows LPE/SBX (Win)
Up to $50,000	6.004 USB LPE (Win/Mac)	8.001 Antivirus RCE (Win)			5.007 WinZip RCE (Win)	5.008 tar RCE (Linux)	6.005 macOS LPE/SBX (Mac)	6.006 Linux LPE (Linux)	6.007 BSD LPE (BSD)
Up to $10,000	9.001 Routers RCE	8.002 Antivirus LPE (Win)	7.005 phpBB RCE (Linux)	7.006 vBulletin RCE (Linux)	7.007 MyBB RCE (Linux)	7.008 Joomla RCE (Linux)	7.009 Drupal RCE (Linux)	7.010 Roundcube RCE (Linux)	7.011 Horde RCE (Linux)

* *All payouts are subject to change or cancellation without notice. All trademarks are the property of their respective owners.*

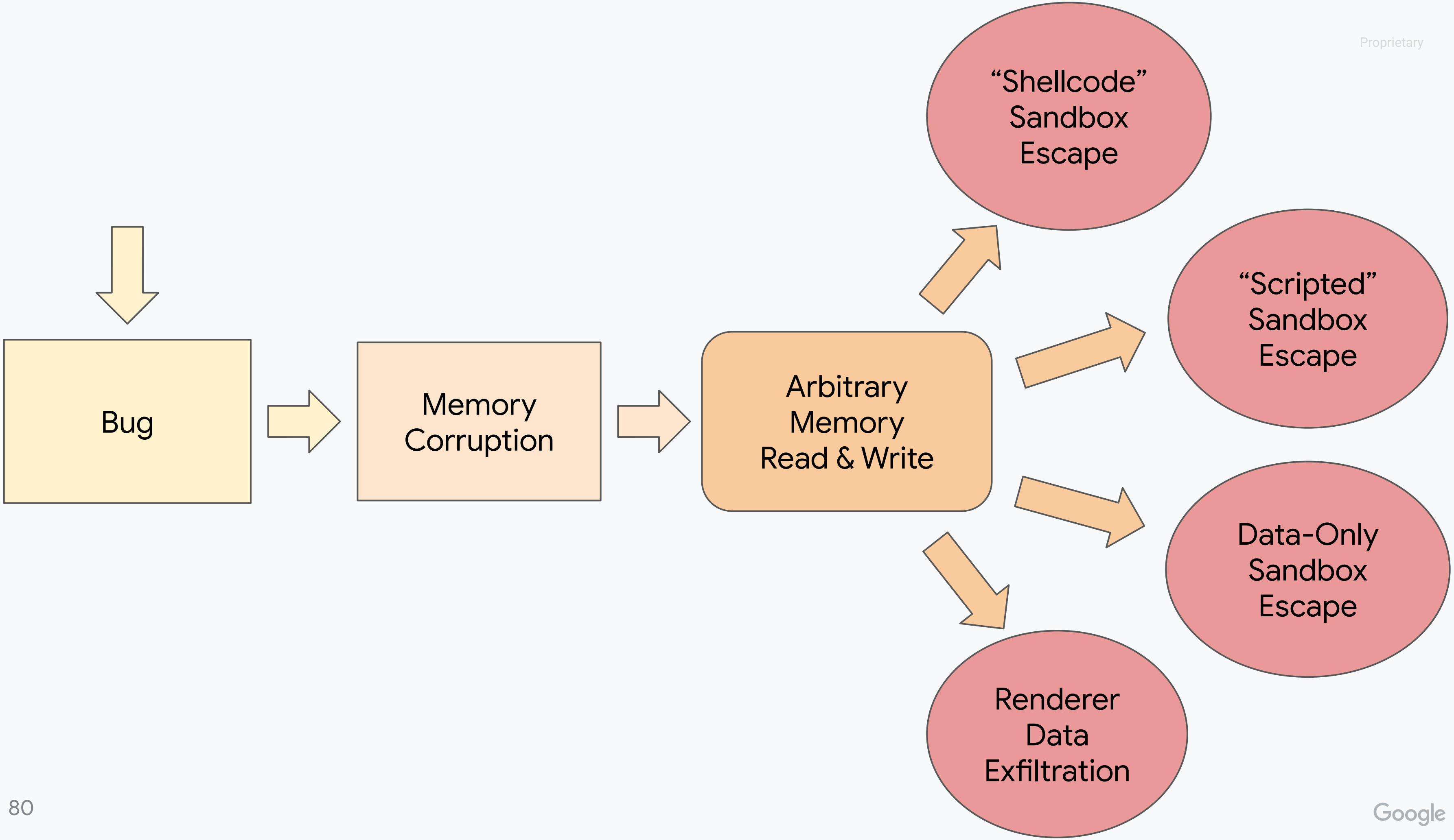

ZERODIUM Payouts for Mobiles*
FCP: Full Chain with Persistence
RCE: Remote Code Execution
LPE: Local Privilege Escalation
SBX: Sandbox Escape or Bypass
iOS
Android
Any OS
Up to $2,500,000
Up to $2,000,000
Up to $1,500,000
Up to $1,000,000
Up to $500,000
Up to $200,000
Up to $100,000
1.001 Android FCP Zero Click Android
1.002 iOS FCP Zero Click iOS
2.001 WhatsApp RCE+LPE Zero Click iOS/Android
2.002 iMessage RCE+LPE Zero Click iOS
2.003 WhatsApp RCE+LPE iOS/Android
2.004 SMS/MMS RCE+LPE iOS/Android
3.001 Persistence iOS
2.005 WeChat RCE+LPE iOS/Android
2.006 iMessage RCE+LPE iOS
2.007 FB Messenger RCE+LPE iOS/Android
2.008 Signal RCE+LPE iOS/Android
2.009 Telegram RCE+LPE iOS/Android
2.010 Email App RCE+LPE iOS/Android
4.001 Chrome RCE+LPE Android
4.002 Safari RCE+LPE iOS
5.001 Baseband RCE+LPE iOS/Android
6.001 LPE to Kernel/Root iOS/Android
2.011 Media Files RCE+LPE iOS/Android
2.012 Documents RCE+LPE iOS/Android
4.003 SBX for Chrome Android
4.004 Chrome RCE w/o SBX Android
4.005 SBX for Safari iOS
4.006 Safari RCE w/o SBX iOS
7.001 Code Signing Bypass iOS/Android
5.002 WiFi RCE iOS/Android
5.003 RCE via MitM iOS/Android
6.002 LPE to System Android
8.001 Information Disclosure iOS/Android
8.002 [k]ASLR Bypass iOS/Android
9.001 PIN Bypass Android
9.002 Passcode Bypass iOS
9.003 Touch ID Bypass iOS
* All payouts are subject to change or cancellation without notice. All trademarks are the property of their respective owners.
2019/09 © zerodium.com

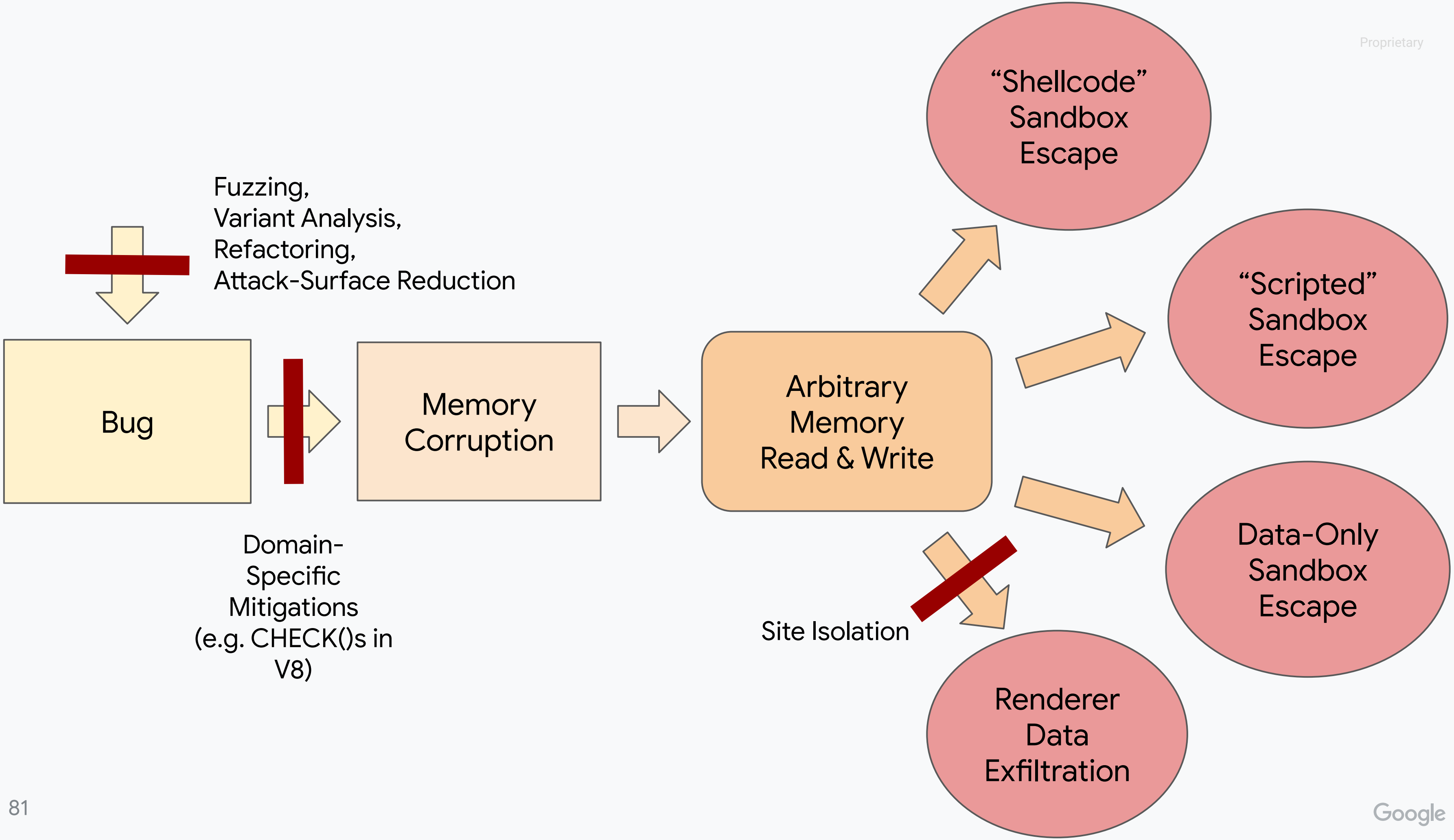
V8's Fundamental Problem

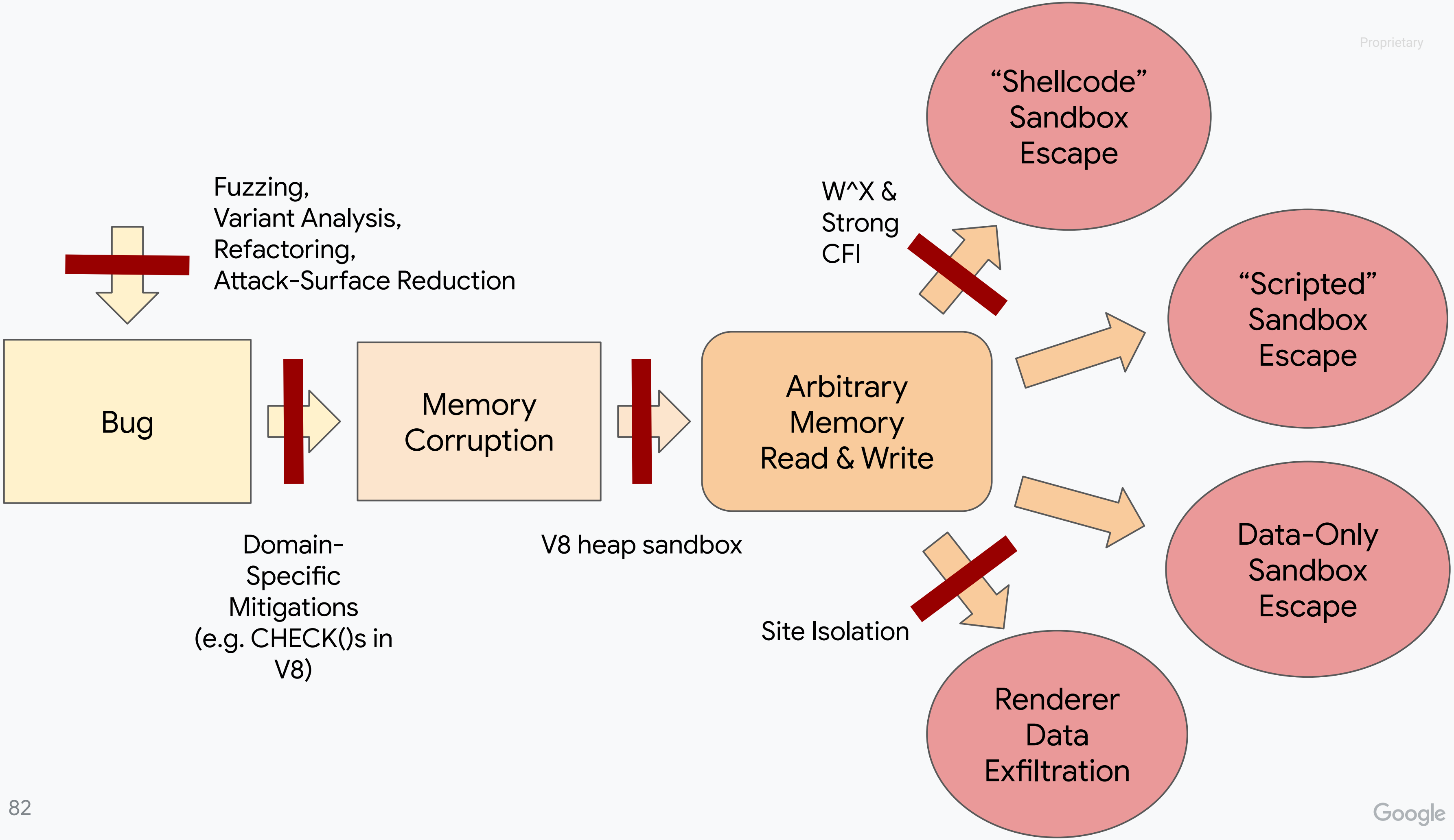
JIT bugs are essentially 2nd order vulnerabilities

Top-tier
compiler
TurboFan

- Root cause is a **logic issue in the compiler** / runtime environment
- ... which is then exploited to **generate vulnerable machine code**
- ... which can then be exploited for arbitrary **memory corruption at runtime.**
- Often cannot even be mitigated with latest hardware features (e.g. CFI).


Bug
Memory Corruption
Arbitrary Memory Read & Write
“Shellcode” Sandbox Escape
“Scripted” Sandbox Escape
Data-Only Sandbox Escape
Renderer Data Exfiltration


Fuzzing,
Variant Analysis,
Refactoring,
Attack-Surface Reduction
Bug
Memory Corruption
Domain-Specific Mitigations (e.g. CHECK()s in V8)
Arbitrary Memory Read & Write
"Shellcode" Sandbox Escape
"Scripted" Sandbox Escape
Data-Only Sandbox Escape
Site Isolation
Renderer Data Exfiltration


Fuzzing,
Variant Analysis,
Refactoring,
Attack-Surface Reduction
Bug
Memory
Corruption
Arbitrary
Memory
Read & Write
Domain-
Specific
Mitigations
(e.g. CHECK()s in
V8)
V8 heap sandbox
W^X &
Strong
CFI
Site Isolation
“Shellcode”
Sandbox
Escape
“Scripted”
Sandbox
Escape
Data-Only
Sandbox
Escape
Renderer
Data
Exfiltration

Summary

https://v8.dev

- Modern JavaScript / WebAssembly engines are complex beasts!
  - Many tiers with various levels of optimizations
  - Necessary given the complex trade-offs
- Basic terms and general techniques:
  - Just-in-time compilers, baseline vs. top-tier
  - Speculative optimizations, deoptimization
  - Unboxed representations, pointer compression
- Many more interesting topics & challenges!
  - Optimization passes: inlining, GVN, escape analysis, regalloc, …
  - Garbage collection, multi-threading, …

Q&A

https://v8.dev

- Modern JavaScript / WebAssembly engines are complex beasts!
    - Many tiers with various levels of optimizations
    - Necessary given the complex trade-offs
- Basic terms and general techniques:
    - Just-in-time compilers, baseline vs. top-tier
    - Speculative optimizations, deoptimization
    - Unboxed representations, pointer compression
- Many more interesting topics & challenges!
    - Optimization passes: inlining, GVN, escape analysis, regalloc, …
    - Garbage collection, multi-threading, …